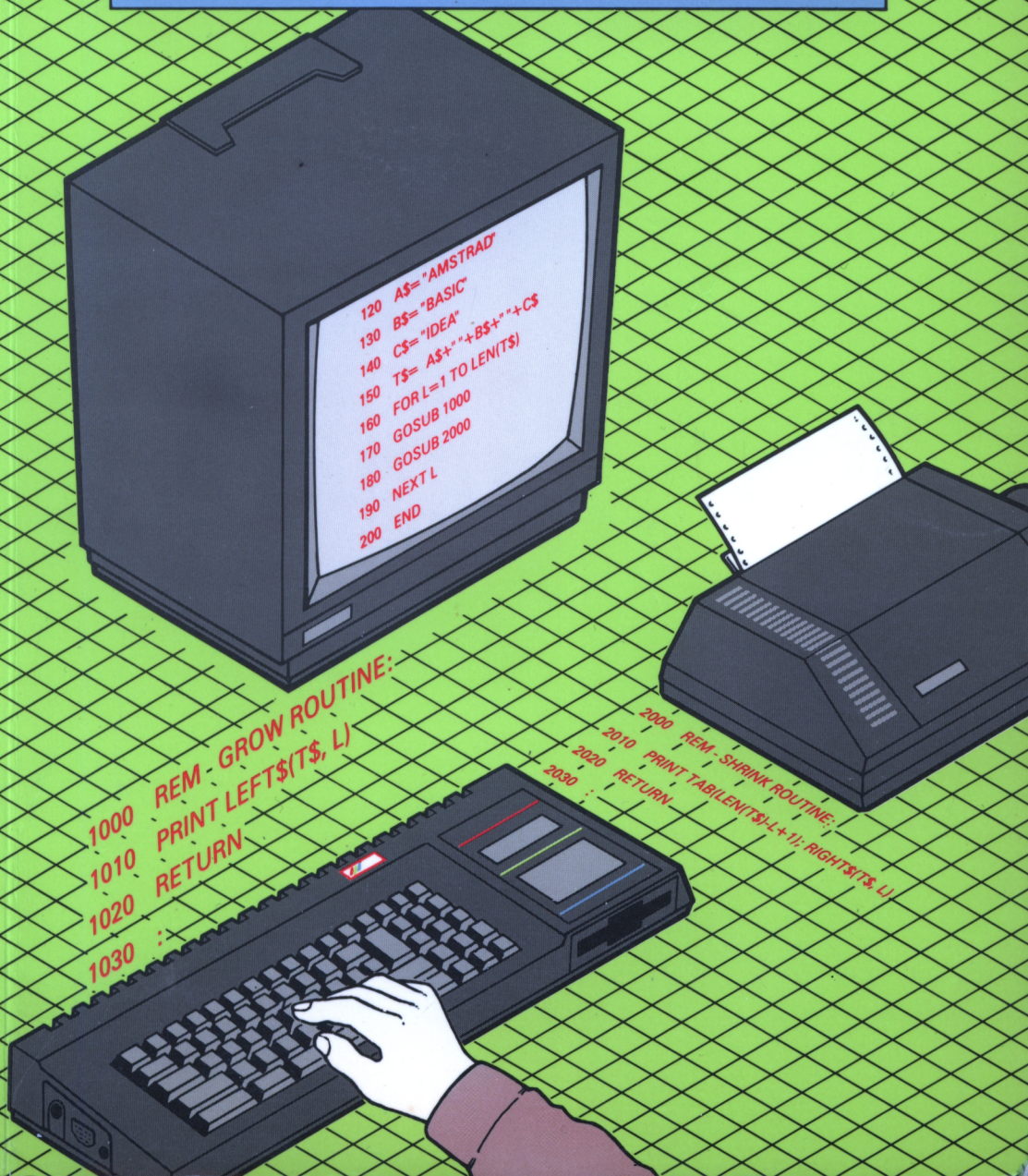
Amstrad

The BASIC Idea

Richard Forsyth and Brian Morris



The AMSTRAD BASIC Idea

The AMSTRAD BASIC Idea

RICHARD FORSYTH
and
BRIAN MORRIS

LONDON NEW YORK
Chapman and Hall/Methuen

First published in 1986 by
Chapman and Hall Ltd/Methuen London Ltd
11 New Fetter Lane, London EC4P 4EE

Printed in Great Britain by
J. W. Arrowsmith Ltd, Bristol

ISBN 0 412 28070 1

British Library Cataloguing in Publication Data

Forsyth, Richard
The Amstrad BASIC idea.
1. Amstrad Microcomputer—Programming
I. Title II. Morris, Brian, 1947–
005.2'6 QA76.8.A4

ISBN 0–412–28070–1

Contents

Preface

Just as the Jumbo jet by getting larger (and cheaper) opened up a new vista of exotic destinations to the ordinary holidaymaker, so the microcomputer by getting smaller (and cheaper) has opened up a whole new realm of computer applications.

This is a guidebook to that remarkable territory. It will teach you enough of the local language to turn you from a package tourist into an independent explorer. The language is Basic. It isn't the only programming language or even the best, but it's certainly the most popular; and it is very easy to learn.

Our book is aimed chiefly at users of the Amstrad CPC 6128 computer, which has set a new standard of price/performance for small systems. All the programs listed have been run on a CPC 6128. However, practically everything in the book is also relevant to CPC 664 owners; and apart from Chapter 7 and Appendices C and D this applies to CPC 464 owners too. This is because the Amstrad range of personal computers (CPC 464, CPC 664 and CPC 6128) all use the same version of Basic, called Locomotive Basic. Even users of the Amstrad PCW 8256 business machine, which provides a version of Basic written by the same software team (known as 'Mallard Basic'), will find much that is applicable to their needs in this book.

In order to widen the generality of the book we have concentrated first and foremost on Basic programming techniques and only secondly on how to make the Amstrad 'sit up and beg' – though of course there are quite a few hints on that sort of thing in later chapters. You will also find a wealth of illustrative programs, many of them more advanced than typical textbook examples, which we believe are important in assisting your progress. (These are catalogued in the index of example programs).

We hope that this book will enable you to use Amstrad Basic to solve realistic problems, and to have some fun doing so. A further objective is to show you how to use modern methods of program design – in other words to profit from some of the lessons learned by computer scientists in the twenty years since Basic was first devised.

If readers have any comments or suggestions for improvement, we will be pleased to consider them for future editions.

December 1985

RICHARD FORSYTH
BRIAN MORRIS

1

Getting started

GENERAL INTRODUCTION

Basic is not only one of the simplest computer languages, it is also one of the most vital. It has developed, during its short lifetime, like a living organism.

Since 1964, when Professors Kemeny and Kurtz planted its roots at Dartmouth College, New Hampshire, it has spread like a hardy weed, thrusting out shoots in all directions. Meanwhile other programming languages, far more elegant in concept, have thrived only in the hot-house atmosphere of the research laboratories where they were born.

The feature which gives Basic its vitality is its responsiveness to change. It has adapted to a changing world. Many new facilities have been grafted on, until today it is hardly recognizable as the 'Beginner's All-purpose Symbolic Instruction Code' of 20 years ago. This flexibility is its greatest strength, but at the same time a major weakness.

In the fast-changing world of computing, to stand still is to die; but the price paid for Basic's flexibility is a lack of standardization. There are many different and incompatible varieties of Basic in existence. A standard form was defined by the American National Standards Institute, but it has been almost entirely ignored by the computer industry.

Fortunately, all dialects of Basic share a distinctive family resemblance. This means that the reader will find it comparatively easy, having mastered one version of the language, to learn another.

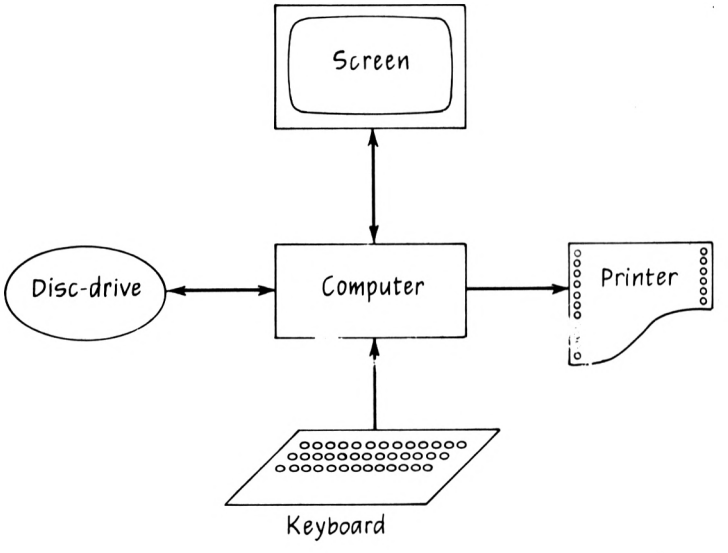
In this book we concentrate on the Basic used by the Amstrad range of microcomputers. This is known as 'Locomotive Basic' after the software house that implemented it. The Amstrad CPC 464 and CPC 6128 computers have recently taken the market by storm, offering a new level of price/performance in personal computing. These remarkable machines use Locomotive Basic. It is a 'plain vanilla' version of the language – well-proven but lacking such modern facilities as procedures with local variables. Its advantage for the beginner is that it does not contain any idiosyncratic constructions that have to be unlearned later. Its disadvantage is that it makes some of the principles of structured programming harder to apply than they should be.

It cannot be ignored, however, and if you have purchased an Amstrad computer you will not get the full benefit of the machine without mastering Locomotive Basic. So let's get started on that task.

1.1 Your computer

You should be sitting comfortably in front of your personal computer. In logical terms it is arranged as depicted in Fig. 1.1. We will assume that you have worked out how to set it up, connect all the leads into the right sockets and switch on. (This is covered in the User Instructions, Chapter 1.) If you cannot afford one yet, you may not have the *printer*. If you have a CPC464, you will have a *cassette* player built-in instead of the *disc-drive* unit. But the general layout will be similar.

Fig. 1.1 Amstrad home computer components



The *screen* is the primary output device. It is a colour or a monochrome monitor, depending on the price you paid. It is the computer's chief means of communication with you, the user.

The Amstrad *keyboard* unit contains the *computer* itself. Some machines have a separate 'system box' which means that the two are detachable; and, logically speaking, it is a good idea to be aware that the two items are distinct. The *keyboard* itself contains the letter keys, the numerals 0 to 9,

many punctuation marks, and several other keys whose use we will describe later. It is your main line of communication to the *computer*.

The *computer* consists of the CPU (Central Processiing Unit) which is a single chip that actually manipulates the data and instructions when programs are executed, the ROM (Read-Only Memory) where permanently resident programs such as the Basic interpreter are held, and some RAM (Random Access Memory) where programs and data are held during execution. Information in RAM may be modified; information in ROM is unalterable. The Amstrad machines use the Zilog Z-80 processor as their CPUs.

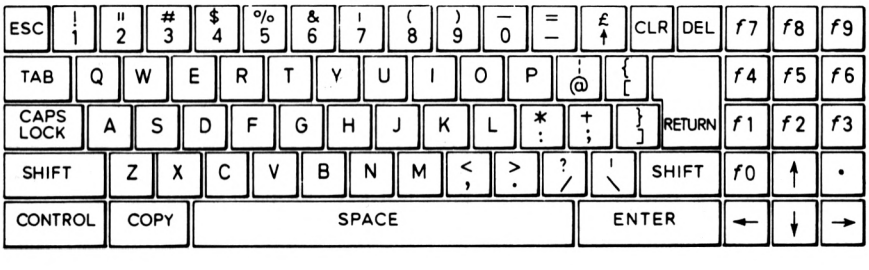
The *disc-drive* is for 3-inch so-called 'floppy' discs. Disc are used for backing storage. Information is recorded on them magnetically and this enables them to hold programs and data when the computer is switched off. Most of this book (with the exception of Chapter 7) applies also to owners of the CPC 464 who do not have a *disc-drive* but rely instead on an integral *cassette* unit. This is a specially modified audio recorder/player on which information can be recorded as tone bursts and later played back. Either system provides for the long-term storage of programs and data, but discs are quicker and more reliable.

The *printer*, which is not supplied when you buy the computer, is for 'hard copy' as it is called in the trade – i.e. printed listings of programs and output. Although we are said to be entering a video age, the human need for seeing things on paper in black and white (and occasionally red, green and blue) remains. Some printers can be made to copy graphical displays too, as patterns of dots. The more programming you do, the more keenly you will feel the need for a *printer*. Fortunately there is a wide range of inexpensive products to choose from.

1.2 The keyboard

The keyboard, as we have said, is your chief means of communicating with the computer. You will have to familiarize yourself with its layout. A diagram is shown in Fig. 1.2.

Fig. 1.2 Keyboard layout



You will notice that the alphabet is arranged in the familiar QWERTY pattern, as on a typewriter. Letters and numbers should present no difficulty, nor should the punctuation signs; but there are also some special keys whose usage may not be obvious. The most important of these are as follows.

Special keys	
CAPS LOCK	Locks the letters into upper case, without affecting other keys (unlike SHIFT). A second CAPS LOCK cancels the first.
CLR	Used to delete the character under the cursor during editing.
COPY	Used in editing: copies the character at which the cursor is positioned.
CONTROL	Is held down while another key is being pressed to generate various special control characters.
DEL	Erases the last character typed in. Pressing DEL *n* times deletes the last *n* characters typed.
ESC	Interrupts a running program. ESC is a kind of emergency exit; press it once to suspend an action; press it twice to halt completely.
RETURN	Causes a carriage-return and line-feed. It must be used to terminate every input line. ENTER has the equivalent effect.
SHIFT	Activates the upper punctuation symbols above the numeric keys and capital letters as long as it is held down.
Shift lock	This is actually achieved by holding down CONTROL and CAPS LOCK simultaneously. Upper case and shifted symbols remain selected until unlocked by the next depression of CONTROL and CAPS LOCK.
TAB	Tabs across to next tabulation setting.
↑	Moves the cursor upwards.
↓	Moves the cursor downwards.
←	Moves the cursor one space left.
→	Moves the cursor one space right.

On the Amstrad all keys have 'auto-repeat'. This means that if you hold them down long enough (about half a second) they start repeating till released. There are also ten *function keys*, labelled f0 to f9. These will be explained later, as will the use of the editing cursor.

Most computer keyboards adopt a similar arrangement, with the exception of function keys (which are often missing) and in some cases cursor control keys.

It is essential for the beginner to get used to the keyboard, and the main requirement for this is practice. There is no need to be an ace typist, but on the other hand if it takes several seconds to hunt for every character then you will not find your computer sessions very productive, or enjoyable.

Notice particularly that the number 1 is different from the letter I and that the number 0 is different from the letter O. The computer is very fussy about these distinctions: if you type one when you mean the other, it will cause problems. Unfortunately some keyboards and many printers do not mark the difference clearly, so take note of where they are placed. You are advised to adopt the following handwriting conventions when preparing your own programs on paper: it could save quite a lot of trouble when typing them into the computer.

Number	Letter
Ø	0
1	I
2	Z
5	S

1.3 The Basic environment

When you use the computer, you are interacting not with a naked machine, but with some very sophisticated software – namely, the Amstrad Operating System (AMSDOS) and the Locomotive Basic interpreter, which are present in ROM. These are programs which convert from terms the user can understand (numbers, commands, character strings, program lines, etc.) to data the machine can handle, i.e. binary code. It may surprise you that the computer cannot execute Basic instructions directly. A program to interpret them is one part of the Basic system. Another part is an editor which allows you to amend programs in memory by moving the cursor around the screen and retyping the incorrect portions. Together this collection of software forms a programming environment.

The fact that Basic is not just a language, but a complete programming environment, is the secret of its phenomenal success; and the user needs to be comfortable with the system as a whole, not just the language, in order to achieve good results with it.

1.3.1. Using the Basic system

When using Basic, you can be in one of two phases:

(1) Input – when Basic is accepting lines into memory and obeying commands;

(2) Execution – when Basic is interpreting and obeying the program currently in memory.

During the input phase (1) Basic regards any line starting with a number as a line to be added, at the point indicated by its number, to the program currently being built up. Any line not beginning with a number is taken as a command. If Basic cannot recognize a command it simply complains, so no damage is done by accidentally typing a nonsense command; but if you type a line number followed by garbage, that line will be faithfully added to the contents of memory and will not cause an error until you try executing it. (Some Basics translate each statement as it is entered, but Amstrad Basic waits till you try to execute a program before checking for errors.)

During the execution phase (2) Basic tries to carry out the program currently in memory.

To get from phase (1) to phase (2), type the RUN command. To force exit from phase (2) and return to phase (1), press the ESCAPE key twice. Normally Basic returns to phase (1) after reaching the end of the program or after an error, so ESC is only necessary if it gets stuck due to faulty programming.

We use the term 'statement' for a line in a program and 'command' for an unnumbered line, obeyed immediately, which tells Basic what to do next. We use 'instruction' to cover both commands and statements. Many Basic instructions, such as PRINT, may be either. For instance,

```
PRINT "Richard"
```

is a direct command, which causes

```
Richard
```

to be put on the screen as soon as RETURN is pressed; while

```
100 PRINT "Richard"
```

is a program statement, which will cause the same output but not until line 100 is eventually executed.

You can enter a Basic program by typing in numbered statements in any order that suits you – thus building it up line by line. Alternatively it can be recalled from disc (or cassette tape) with the LOAD command if it has been saved previously. Once loaded or entered a program can be run by giving the RUN command.

A short list of essential commands is given below.

Command	Meaning
LIST	Lists current program on the screen.
LOAD prog	Loads a named program from backing store.
NEW	Clears memory ready for input of a fresh program from the keyboard.
RUN	Executes the program currently in memory.
SAVE prog	Saves the current program on disc or cassette under the name specified.
CAT	Gives a catalogue of the saved entries on a disc.

In this list 'prog' stands for the quoted name of a program. The name may be up to eight characters long. Thus

LOAD "JOKE"

tells the system to search for JOKE on disc (or cassette) and bring it into memory if found – overwriting anything currently there. Of course, this implies that

SAVE "JOKE"

was performed on some previous occasion.

Further Basic commands will be presented later. These are enough for the novice to start with.

1.3.2. A sample session

Here is the printout from a complete session to give you an idea of what a Basic program looks like in action. Do not be dismayed if you cannot yet understand how it works: the instructions it uses are explained in the next two chapters. For the moment, all you need to notice is that a Basic program consists of a series of statements each preceded by a line number.

Listing 1.1 Probability calculations

```
10 REM ********************************
20 REM ** Listing 1.1 :              **
30 REM ** PROBABILITY CALCULATIONS  **
40 REM ********************************
100 PRINT: PRINT "Probability Calculator :"
110 PRINT "Please give odds as two numbers;"
120 PRINT "e.g. 3,1 for 3-to-1 etc."
130 PRINT "Use 0,0 to end input.": PRINT
140 ZONE 4
150 n=0
```

```
160 tp=0
170 a=1: f=1
190 REM -- Main Loop:
200 WHILE a>0 AND f>0
210   n=n+1
220   PRINT "Runner ";n;" name is ";
230   INPUT name$
240   PRINT "Odds against are ";
250   INPUT a,f
260   IF a<=0 OR f<=0 THEN GOTO 330 : REM quit
270   prob = f/(a+f)
280   tp = tp + prob
290   PRINT "For runner no. ";n;", ";name$
300   PRINT USING "probability is : ##.####"; prob
310   PRINT
330   WEND
350 n=n-1
360 PRINT
370 PRINT "Stake returned is ";100/tp;"%"
380 PRINT "Bookies' mark-up: ";100*(tp-1);"%"
390 IF tp<1 THEN PRINT "You're kidding!";CHR$(7)
400 END
RUN

Probability Calculator :
Please give odds as two numbers;
e.g. 3,1 for 3-to-1 etc.
Use 0,0 to end input.

Runner  1  name is ? home
Odds against it are ?  15 , 8
For runner no.  1 , home
probability is :  0.3478

Runner  2  name is ? draw
Odds against it are ?  5 , 2
For runner no.  2 , draw
probability is :  0.2857

Runner  3  name is ? away
Odds against it are ?  1 , 1
For runner no.  3 , away
probability is :  0.5000

Runner  4  name is ?
Odds against it are ?  0 , 0

Stake returned is  88.2191781 %
Bookies' mark-up:  13.3540373 %
```

In this sample session a program called ODDS is loaded into working memory, then listed and run.

This program calculates the winning probability of each runner in a race corresponding to the bookmaker's odds. These probabilities are totalled to work out the margin of the bookies over the punters (or the other way round in the unlikely event that they have slipped up). This is calculated simply by adding up all the probabilities. In a fair race they will total 1 (100%). When they add up to more than one, as is normal, the bookmaker has an advantage. (Well, he isn't running a charity.)

In the example shown, the bookmaker is liable to return just over 88% of the money wagered. This is about average for betting on the turf; but a considerably better return than in the 'football pools'.

The program is actually designed for horse races, but the sample output shown here results from testing it on a soccer match. The odds were taken from a fixed-odds football coupon for 17 August 1985 on a game between Leicester City and Everton, and were as follows:

Home win	15–8
Draw	5–2
Away win	1–1 (evens)

As it happens, the home side won that game, proving yet again that favourites cannot always be relied on.

Note that we convert from odds against an event **(A/F)** to the probability of that event occurring **(P)** with the formula

P = F/(F +A)

which is used on line 270 of the program. However, do not worry if much of it seems incomprehensible, or indeed if gambling bores you. All we want to do at this stage is to show you what a fairly short (but potentially useful) program looks like.

1.4 The next step

The fastest way to learn to use Basic is by practice. Because of its interactive nature, a Basic system acts somewhat like a teaching machine. But before rushing wildly to the keyboard, read this section. It discusses some common pitfalls encountered by beginners.

1.4.1 Practical details

Certain skills which appear to have nothing to do with programming must be mastered in order to utilize the computer. Before going any further you should be reasonably confident that you know how to do the following things. (Note that from now on, for brevity, we will stop referring to cassette and assume you have a disc unit. CPC 464 owners please excuse us: almost everything we say still applies to users of cassette-based systems.)

(1) Turn the machine on and connect all the components correctly!

(2) Call in a stored program from disc.

(3) Execute a program that has been loaded.

(4) List a program.

(5) Save a program on disc for use at a later date.

(6) Halt a running program when something goes wrong.

(7) Erase one or more characters on the line you are currently typing, or erase the whole line, before transmitting it to the computer.

(8) Recognize the most common error messages and what they signify.

In addition you ought to find out how to format a disc. Formatting is the process that takes a blank disc as it comes from the manufacturer and writes control information on it so that AMSDOS can use it for storing program files. CPC 6128 owners will find details on page 1–38 forwards of the User Instructions. Owners of cassette-only systems need not bother with formatting.

These necessary but arbitrary points are those which tend to cause most confusion and annoyance among people who have never used a computer before. The difficulties of learning Basic are minor by comparison.

It is important to get the feel of conversational computing; to get to know the keyboard and the effects of the various special keys; to master the essential procedures such as saving and loading; and to gain a sense of competence in using the computer, before going on to your own programming projects. If the system is strange to you when you come to try out your own programs, you will experience needless frustration. A good way to practise these basic skills is to type in short programs from computer magazines, or from this book, and try to make them run. Working out why they do not (not first time, anyway) can be an instructive pastime.

If you stumble over the first steps, you may be tempted to say 'oh well, I'm not a computer person' and quit. Rather than give up so easily, try to find someone who is a 'computer person' to show you the ropes. There are plenty of them about.

To err is human; to learn from our mistakes is human too. Even the experts were incompetents once. Some of them can still remember it.

1.4.2. Program design

Computers can be used, and misused. Two pitfalls to avoid when you start interacting with the computer are: firstly, sitting down with no idea of what you plan to do; and secondly, attempting something too complex too soon, and becoming discouraged. On the one hand your mind should not be blank when you confront the computer, and on the other hand you should not attempt anything over-ambitious (and fail) in the early stages. If you do, your home-computing career will be a short one.

Remember also that programming is a problem-solving exercise, not purely a matter of coding. So do not try to make it up as you go along. Analyse

your problem first, then write down its solution. The golden rule is: put your program on paper before putting it into the computer.

Over the years certain principles of good program design have been discovered. This theme will be taken up again at greater length in Chapter 9. Here it suffices to say that many beginners ignore these principles, as do many experienced programmers, through excessive haste. Time spent in thought beforehand is handsomely repaid in time saved later trying to fix mistakes that stem from poor program design.

In this game impatience is the gravest sin. So

THINK before you CODE before you TYPE.

2

Basic Basic

ELEMENTARY PROGRAMMING

In order to make the computer do calculations for you, it is necessary to specify exactly the sequence of operations it must perform. In other words, it has to be given a *program* (a list of instructions) in a language it can understand. One such language is Basic. A program in Basic consists of an ordered set of *statements* each one of which instructs the computer to carry out a single step in the computation. These statements are English keywords such as LET and PRINT, sometimes followed by mathematical expressions and always preceded by a line number.

Thus, to apply the computer to your particular problem, using Basic, you must first break the problem down into a detailed sequence of small steps, then write the Basic instructions to carry out those steps in the correct order. This will be a Basic program for that task, which you can type in at the keyboard and test-run, amending it where it fails to work, until you have a successfully running version that can be saved for subsequent re-use. This process is illustrated in Fig. 2.1 below.

This is inevitably a simplified outline, but it does capture two important points: firstly, that the business of programming is a process, comprising several more or less well-defined stages; and secondly, that we can describe the process in a hierarchical fashion – as having levels. The higher levels are more general, the lower levels more specific. This is a key feature of all modern approaches to programming.

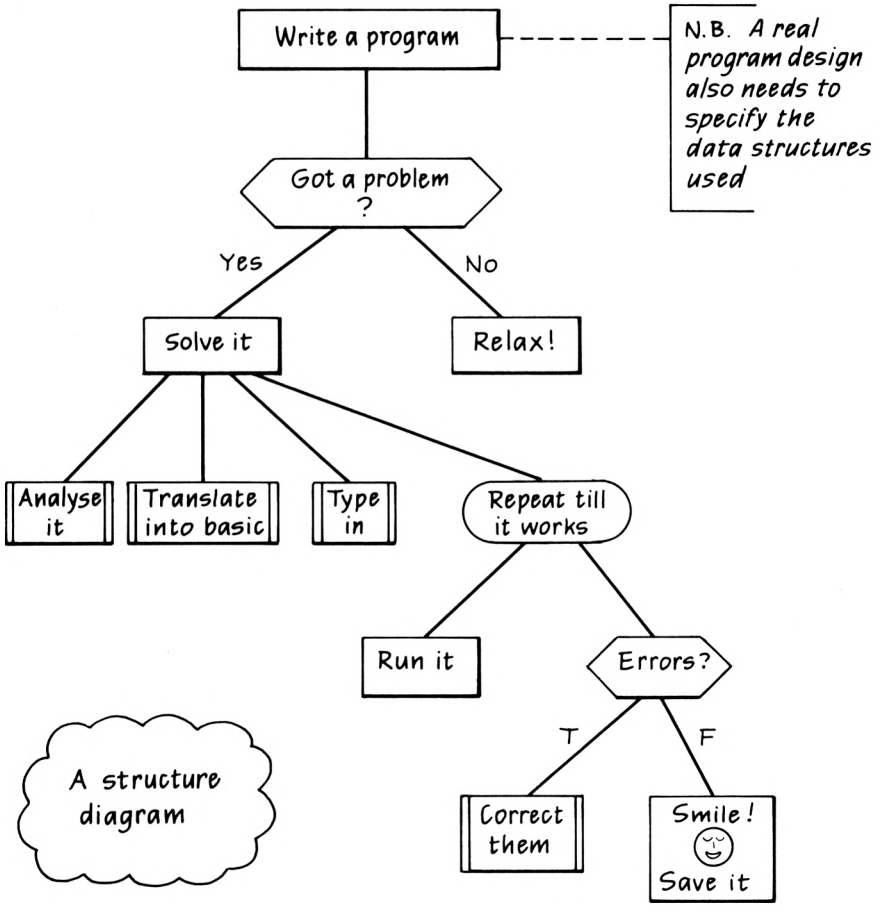
Notice also that we have presented the process in diagrammatic form, as a *structure diagram*. This visual method of representing processes can be extremely helpful during problem analysis, and we shall use it again later in the book. (The technique shown here is an adaptation of that used by Roy Atherton in *Structured Programming with Comal*, Ellis Horwood, 1982).

We use four kinds of boxes to draw structure diagrams. See Fig. 2.2.

A *processing action* is put in a rectangular box. Simple actions need not be, or cannot be, further subdivided.

A *subprocess* is a process which is complex enough to warrant its own

Fig. 2.1 The programming process

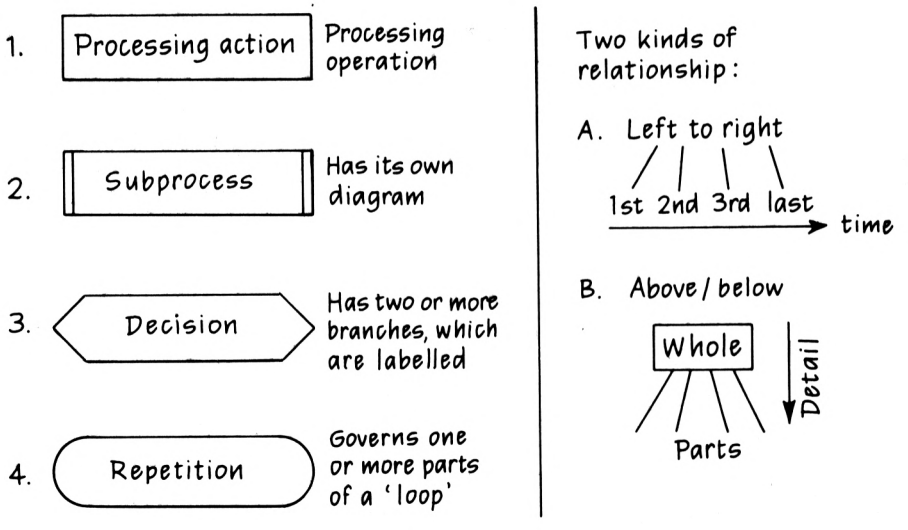


description, in terms of another structure diagram, on a separate page. This allows us to avoid getting bogged down in details.

A *decision* governs two or more branches. These are subsections of the whole, only one of which will actually be performed – depending on conditions tested when the decision is made. Each line descending from a decision box is labelled (e.g. Yes/No, True/False, T/F etc.) to show which choice it represents.

A *repetition* box governs a subsection of the overall process which is repeated as long as, or until, some condition holds. In computing parlance it is known as a 'loop'.

Fig. 2.2 Structure diagram symbols



Decisions and loops are what give computers their great power, as we shall see.

The lines connecting the boxes show dependency (downwards). Temporal sequence is shown by horizontal position (across). A box *below* another one is a *part* of the one above; a box to the *right* of another is carried out *after* the box to its left.

Structure diagrams are not only (like flowcharts) useful for presenting the logic of a solution in an eye-catching manner they can also (unlike flowcharts) help in finding the solution in the first place. Moreover, you can convert a structure diagram into a well-structured program by following a simple set of rules: essentially you turn the thing on its side and walk round the tree translating the contents of each box into their Basic equivalent. (We shall have more to say on this in Chapter 9.)

We could use structure diagrams to elucidate the logic of all sorts of sequential processes – e.g. cooking a spaghetti bolognese – but we really want them for constructing computer programs. We cannot do that until we have a kit of parts from which programs can be put together.

Accordingly, this chapter introduces seven essential Basic statements. They form a foundation on which you can build as you progress. There are programming tasks which cannot be carried out using only these seven statements; but there is a surprisingly wide variety of tasks which can be. Furthermore, it would be very hard to find a Basic program of more than a few lines that did not use most, or all, of them.

But first, a little background on how a computer system stores information. Before we can think clearly about a program, we have to think about the data it will use.

2.1 Data types

Computers are information processors. When you learn to use a computer, you are learning, among other things, how it handles information.

At the machine level all data is encoded in terms of binary digits or 'bits' which may be 0 or 1.

For convenience, most computers group information into 'bytes' of 8 bits (or 'words' of anything from 16 to 64 bits) and deal with a byte or a word as a unit.

It is important to realize that the meaning of a group of bits is not fixed: it all depends on what you do with it. A byte has no absolute value in itself. Its bit pattern merely selects one of a set of possible alternatives. It is an answer – which only makes sense when you know the question.

With 8 bits there are 256 possibilities. They could be used to represent numbers from 0 to 255, or from −128 to +127, or one of a repertoire of 256 characters, or a machine code instruction, or a line of dots on the screen . . . or whatever you choose. The machine, or the programmer, assigns a different meaning to each one of those 256 states.

Some meanings, however, are more likely than others; and these tend to be pre-assigned in various ways. For example:

00101010	**'means':**
42	as a decimal number,
2A	as a hexadecimal number,
LD HL,	as a machine instruction (load the 16-bit HL register with the contents of a given memory address),
'*'	as an ASCII character code,
..X.X.X.	as a pattern of dots on the screen (with X for on and . for off), and so on.

However, people – unlike computers – are not built to handle bit patterns. We find them confusing. We would rather deal with numbers or letters or other familiar symbols. One of the many things that Basic does for you is to convert from humanly meaningful information into bit patterns and back again automatically.

Locomotive Basic provides three primary data types, from which others can be constructed. These are: *integer* or whole numbers, *floating-point* or fractional numbers, and character *strings*. For example:

1 is an integer,
1.5 is a floating-point number,
"Two" is a character string.

We shall postpone detailed discussion of strings until Chapter 5, but it is as well to know they are there.

2.2 Arithmetic in Basic

Many people think of computers simply as overgrown calculators – 'number crunchers'. You will find later (for instance, in Chapters 5 and 7) that Basic can be used for other kinds of information processing, but of course numerical calculation is very important and Basic provides extensive facilities for it.

2.2.1 Numeric constants

A numeric constant is written as a series of decimal digits optionally preceded by a sign and optionally followed by an exponent part – the letter E followed by an optional sign (+ or −) followed by one or two digits. Leading zeros are not significant. Examples of valid numeric constants are shown below.

Integral	0	1	1988	−77
Decimal	0.75	1.60934	100.001	−9.87654321
Exponential	1E7	5E−10	−99.99E+9	0.314159265E01

Commas are not allowed within a number, so twelve thousand is 12000 and not 12,000. Nor are spaces permitted, so 12 000 is also wrong.

In exponential notation E stands for 'times ten to the power of' and is used for scaling. Thus 1E7 equals one times ten to the power of seven which is ten million (10000000), and 0.314159265E01 is an approximation to pi (3.14159265). This scientific notation can save a lot of typing: consider the number 1.99E33 (the mass of the sun in grams). Writing it out in full would require 199 and then 31 zeros.

The number after E is known as the 'exponent', hence 'exponential' notation. A positive exponent of *n* effectively shifts the decimal point *n* places to the right; a negative one shifts it *n* places to the left.

You can also use hexadecimal notation to represent integers in Locomotive Basic. In this system numbers are represented to the base 16, and the letters A to F are used after 9 to give a full set of sixteen digits (0 to F, where F=15). Hexadecimal numbers are preceded by an ampersand (&), so &AA is the same as 170 (10×16+10).

Amstrad Basic stores integers as pairs of 8-bit bytes and floating-point

numbers using five bytes. The integers range from −32768 to +32767. The largest floating-point number is about 1.7E+38 (17 followed by 37 zeros) and the smallest approximately 2.9E−39. Anything smaller than that is indistinguishable from zero. Negative floating-point numbers are limited to the same range with opposite sign. If a computed quantity is too big to hold, the error message number 6 "Overflow" is printed and the program halts.

The point to remember is that numbers are restricted to a large but finite range and that the computer's calculations are limited to at most ten decimal digits of precision. Integers have a smaller range but are held exactly as whole numbers. Floating-point values have a larger range but a lesser precision: they are best regarded as approximations. To overcome these limitations on size and accuracy requires extra programming effort. (This imprecision frequently surprises newcomers who have been led to think of computers as symbols of inhuman accuracy.)

2.2.2 Variables

A constant in program represents a fixed value. A value that can alter during program execution is known as a variable.

In Amstrad Basic variables are given names by the programmer which have to begin with a letter. Variable names may contain uppercase and lowercase letters and digits, but may not be the same as a Basic keyword such as PRINT. Thus

EDDY **Eddy** **A1** **Frances**

are all legal variable names but

1A (starts with a digit)
2% (likewise)
37 (is a number)
Next (**NEXT** is a keyword)

could not be.

Note that in Amstrad Basic upper and lower case variable names are equivalent; so **RICH**, **Rich**, **rich** and indeed **rIcH** all refer to the same variable.

Variable names followed immediately by a percent (%) sign, such as **COUNTER%**, are for integer values only. You can also append an exclamation mark (!) to variable names to indicate that they are for floating-point numbers, but this is the default case anyway. Integer numbers take up less storage space and are processed more quickly than floating-point numbers. Therefore there is considerable advantage in using integer variables for those parts of a computation where whole numbers are being manipulated.

It is also possible to use the DEFINT instruction to declare that variables

starting with certain letters are integers. Thus, for instance, **DEFINT b,c** tells the system that variable names with initial B or C should be regarded as integers.

You can think of a numeric variable as naming a storage cell capable of holding one number, which may change during the course of a computation. The variable's name can be used to denote the value currently held at that location.

Amstrad Basic presets numeric variables to zero, but it is not good practice to rely on this fact (since it is not common to all Basics). Programs look clearer when all variables are explicitly initialized, even when they start at zero.

2.2.3 Arithmetic expressions

Arithmetic expressions in Basic follow normal algebraic conventions as closely as possible, subject to the constraints that all symbols must be on one line and that all operators must be explicit, not implied.

The simplest form of expression is a constant or variable. More complex expressions are built up by linking simpler expressions with operators. Subexpressions may be enclosed in brackets.

The arithmetic operators, in order of precedence, are:

^	exponentiation;
* and /	multiplication and division;
+ and −	addition and subtraction.

The order of precedence is important. When evaluating an expression Basic follows these rules to produce the final result:

(1) Evaluate subexpressions in brackets, if any;

(2) Evaluate any exponentiations, left to right;

(3) Evaluate multiplications and divisions, left to right;

(4) Evaluate additions and subtractions, left to right.

Thus **A + B * 2** means first multiply **B** by **2** then add **A**; whilst **(A + B) * 2** means add **A** and **B** then multiply by **2**. Notice that paired brackets can be used to impose any desired order of evaluation, if the default priorities are not suitable.

Amstrad Basic also has the operators **** and **MOD**, with the same priority as multiply and divide: they are used for whole-number operations. The backslash (****) divides two integers and truncates the result to a whole number. **MOD** yields the remainder. Thus **7 \ 2 = 3** while **7 / 2 = 3.5**; **7 MOD 2 = 1** and **15 MOD 4 = 3**.

The picture is somewhat complicated by the three roles of the plus and

minus signs. The minus sign, for instance, may appear as the sign of a number **(−16)**, as the negation operator **(−X)**, and as the subtraction operator **(16 − X)**. Confusion can be avoided by the judicious insertion of brackets.

The best policy for the beginner is to avoid long complex expressions altogether by breaking them into smaller steps, and to insert brackets wherever ambiguity might arise – even if not strictly necessary.

A few examples will help clarify these rules. (Assume that **A = 1**, **B = −2**, **C = 10** and **Z = 0**.)

Expression	Result
+1	**1**
B	**−2**
−C	**−10**
Z+10	**10**
A − B	**3**
C * 10	**100**
A / C	**0.1**
2/C + A	**1.2**
2 / (C + A)	**0.18181818**
4 * C ∧ 2	**400**
(4*C) ∧ 2	**1600**
B*C/(4−Z) + A	**−4**

As a further illustration, consider what happens when the final example is evaluated.

B * C / (4 − Z) + A	original expression
−2 * 10 / (4 − 0) + 1	compute values of variables
−2 * 10 / 4 + 1	evaluate subexpressions in parentheses
−20 / 4 + 1	leftmost multiplication
−5 + 1	division
−4	addition

Incidentally, spaces are not significant in arithmetic expressions (except after **MOD**), so **A + B − C** means the same as **A+B−C**.

To round off this section, here is a list of illegal expressions together with brief explanations of why they are not valid. A very common mistake is not having the same number of left and right brackets.

Invalid expression	Reason
1 +	nothing after +
INPUT − 1	**INPUT** is a keyword
A.B	multiplication is * not .
7B / 2	**7B** is neither variable nor expression

Invalid expression	Reason
X & Y ı Z	**&** and **ı** not operators
Q//A	double division operator
(A+B)/C)	too many right brackets
J ∧ J − 55E55	constant too large
98.6.	too many decimal points
(Q5 + b	unpaired left bracket
123,456 + 1%	comma not allowed in number
96%	**%** comes after variables, not constants
***£!?**	gibberish
AAAARGHH!!!!	OK, don't panic; we've finished.

2.3 The LET statement

The LET statement is the first, and probably the most important, Basic statement we consider. It assigns a value to a variable. Its general form is

[LET] variable = expression

where the word LET is optional (signified by square brackets). On the left of the equals sign is a variable name, on the right is an arithmetic expression.

It causes the expression to be evaluated and then assigns the resultant value to the variable named. The LET statement, then, has two functions: firstly, to evaluate an expression or formula; and secondly, to alter the value of a variable so that the result is retained. Some examples follow.

```
60   LET A = 1
100  LET N% = 0
200  LET j2 = j2 + 1
400  LET I% = I% \ 2
800  junk = A*B / (C−D)
808  garbage = junk
888  NINE = FOUR + FIVE
```

Line 60 sets variable **A** equal to **1**. Line 100 sets **N%** equal to **0**. Line 200 adds **1** to **j2** by first evaluating **j2+1** then assigning this new value to replace the old value of **j2**. Line 400 also has the same variable (**I%**) on both sides of the assignment, effectively halving it. Lines 800 to 888 show that the word LET can be dropped: any statement not starting with a recognized keyword is taken to be a LET statement.

Newcomers sometimes have trouble with assignments such as **J=J*2** where a variable appears on both sides of the equals sign (as in lines 200 and 400 above). It is not an algebraic equality, but an order to the computer. Once you realize that it has two steps – first compute a value, then store it in a variable – you should no longer find it puzzling.

2.4 Getting down to basics

Six further Basic keywords are now described which, together with LET, form the nucleus of the Basic language. With them you can start to write useful programs.

Each statement in Basic normally occupies its own numbered line, but it is possible to use the colon (:) to separate two or more statements on the same line.

Note: In the model statement formats presented from here onwards, words in CAPITAL letters are keywords and must be typed as they appear. Words in lower-case letters describe constructions and should be replaced by an instance of the right kind, as detailed in the explanatory text. Square brackets [ ] enclose optional parts and curly braces { } enclose portions that may be repeated zero or more times. Other punctuation marks, e.g. commas, should appear as shown.

2.4.1 END

We begin, deliberately, at the END. END terminates the program. Its format is simply

END

and its effect is to inform the Basic system that it has reached the end of the program. In Amstrad Basic (unlike some other versions of Basic) END may occur more than once in a program, indicating several endpoints.

When END is reached execution stops, Basic gives the prompt ("Ready") and the system reverts to its input phase awaiting user commands.

STOP is an alternative keyword with a slightly different effect. It halts execution but leaves the program in a state where it can be resumed by the CONT command. STOP can be useful for interrupting a program at a particular point during testing.

If a program has no END or STOP statement, it can only be halted by pressing ESC (twice), by switching off the power or by 'falling through' the last program line. None of these methods is very elegant.

2.4.2 GOTO

Unless specifically diverted, Basic programs are executed in ascending line-number order. GOTO interrupts normal flow of control and causes transfer to a specified line. Its form is

GOTO line-number

where 'line-number' specifies the next line to be executed which does not

have to be the line after the GOTO itself. Its purpose is to allow branching to another part of the program. Thus in

```
100 REM preparing for take-off:
110 GOTO 200
120 REM over the top!
200 END
```

line 120 is skipped because line 110 instructs the computer to take the next instruction from line 200, which stops the program.

Some Basics allow the destination of a GOTO to be an expression, so that statements like

```
GOTO Jail
```

are legal; but Amstrad Basic insists that the line number must be a constant.

This seemingly inoffensive four-letter word is in fact one of the most notorious statements in computing. It has received a vitriolically bad press – not without reason, since its injudicious use makes programs incomprehensible.

We shall see soon that GOTO can usually be avoided. Meanwhile, treat it like strong liquor: a little can be pleasant; overindulgence will prove harmful.

2.4.3 IF

This statement can divert the normal sequential flow of processing, but only if a given relationship holds. Thus the programmer can test whether conditions have arisen during processing and make the program do different things depending on the outcome of the test.

The IF statement introduces the possibility of decision-making into Basic. It is essential for any non-trivial program. Its form is

```
IF condition THEN action [ELSE action]
```

where 'action' stands for one or more statements (separated by colons if there is more than one, but all on a single line).

What happens is that the statement or statements after THEN (the THEN-clause) are executed if the condition is true; and the statement or statements, if any, after ELSE (the ELSE-clause) are executed, if it is false. The ELSE part is optional. If it is omitted, and the condition is false, the 'action' consists of doing nothing. Thus

```
125 IF A < THEN PRINT "Sub-zero!"
```

prints the phrase

```
Sub-zero!
```

on the screen if and only if variable **A** has a value less than 0. Otherwise it does nothing. While

```
128 IF A > B THEN PRINT A ELSE PRINT B
```

prints either **A**'s or **B**'s value, whichever is greater.

Conditions can be much more complex than the simple comparisons shown above. Amstrad Basic provides relational operators for comparison and logical operators for combining comparisons.

The relational operators are as follows:

<	less than
>	greater than
=	equal to
<=	less than or equal to
>=	greater than or equal to
<>	not equal to

They allow numeric values to be compared. Thus

```
dollars < = 1000
```

is true if the value (i.e. contents) of variable dollars is less than or equal to 1000; while

```
A ∧ 2 + B ∧ 2 = C ∧ 2
```

is only true if **A** squared plus **B** squared equals **C** squared; otherwise it is false.

The logical operators NOT, AND and OR (sometimes called Boolean operators in honour of the nineteenth-century logician George Boole) allow simpler conditions to be combined.

NOT simply inverts the truth of the operand that follows it. Thus

```
NOT 0
```

is true, and

```
NOT (A > B)
```

is equivalent to

```
A <= B
```

in effect.

AND is true if both its operands are true, false otherwise; OR is true if either of its operands is true, false if they are both false.

NOT has the highest priority, AND the next highest priority and OR the lowest priority among the logical operators. All relational operators are equal in priority. They fit in just above the logical operators (and just below all the arithmetic operators) in precedence.

Thus

```
1024  IF A + B > C AND C > = 0 OR J = 0 THEN....
```

means the same as

```
1025  IF (((A+B) > C) AND (C >= 0)) OR (J=0) THEN....
```

where redundant parentheses have been inserted to show which operators bind which operands.

N.B. Amstrad Basic actually interprets the numeric value zero as false and any non-zero value as true, as illustrated by **NOT 0** (above). Consequently it is possible to have variables or arithmetic expressions as conditions, such as

```
2000  IF abnormal% THEN PRINT "error!"
```

though this may sometimes lead to obscurity.

2.4.4 INPUT

This statement calls for values from the keyboard. It is a way to get information from the user into the program while it is running. Its form is

```
INPUT [message;] varlist
```

where 'varlist' stands for a series of one or more variable names (not expressions) separated by commas.

When executed, it causes the computer to print the message (if there is one) then a question mark, and wait for user input. The program will only continue when the user has supplied values for all the variables in the list. Thus

```
INPUT A,B,seed,percent%
```

causes **?** to appear on the screen. Then the computer pauses, expecting the user to type precisely four numeric constants, separated by commas and terminated by RETURN, of which the last must be a whole (not fractional) number. These four numbers will become the new values for **A**, **B**, **seed** and **percent%** respectively.

Take care to supply as many numbers as there are variables in the input list: strange effects can arise from typing 3,4,5 when you meant 3,4.5 and similar small slips.

The optional 'message' is a character string in quotation marks. It is used as a prompt, reminding the user of what is expected. Thus

```
100  INPUT "Give no. of credit items "; Credits%
```

will cause

```
Give no. of credit items ?
```

to appear, and await a value for the integer variable **Credits%**. This is clearly more helpful than the bald

```
100 INPUT Credits%
```

which requires the user to know how the program works.

(If the semicolon after the message is replaced by a comma, the question mark will be suppressed.)

2.4.5 PRINT

This is used for output. Its basic form is

```
PRINT list
```

where 'list' is a sequence of expressions, normally separated by commas or semicolons. The values of the expressions (which can of course simply be constants or variables) are calculated and printed in order on the screen. A new line is begun after each PRINT statement, unless the list of expressions ends with a semicolon. A new line can be forced at any stage by a PRINT statement with no list of expressions.

Two examples follow.

```
48 PRINT 10, A, 10/A, A/10
80 PRINT "VALUE NO. ";J%;" IS" (A-1)*J%
```

The first would cause the line

```
10      20      0.5     2
```

to be printed (assuming **A=20**); and the second would cause

```
VALUE NO. 2 IS      38
```

to appear (assuming **J%=2**).

Items in the PRINT list separated by semicolons are closely packed together while items separated by commas are spaced out, normally in print zones of thirteen character positions. The size of the print zone can be altered by the ZONE instruction. Thus

```
ZONE 8
```

would set the zone width to eight characters. (More details can be found in Chapter 6.)

Some additional points worth noting are the following.

(1) A PRINT statement with no output list emits one blank line.

(2) Ending an output list with a semicolon suppresses the new line normally output at the end of each PRINT, so the next PRINT will add more output immediately after the place where the last one finished.

(3) String constants (i.e. sequences of characters enclosed within double quotation marks) may be items in an output list, as in line 80 above, in which case they are printed just as they appear, without the quotation marks.

(4) Where no ambiguity would arise, the semicolon or comma separator may be omitted between two items, in which case the semicolon is assumed. (For clarity, we do not exploit this short cut in the book: it is not recommended.)

Further information on PRINT formatting can be found in Chapter 6. It can get quite complex, but there is enough here to begin with.

2.4.6 REM

This statement is often regarded as unimportant by the new programmer, and by many experienced programmers; yet some experts believe that at least half the lines in every program should contain REM statements!

REM is simply a means of putting remarks or comments in the body of a program. Its form is

REM text

where the 'text' can be any printable characters. Its purpose is to allow the insertion of explanatory remarks in a program. Programs can be written without REM statements and can run successfully without them, but you will find they are useful when you come to revise a program written earlier by yourself or, worse still, by someone else.

Usually, it only takes a couple of weeks to forget what the variables were for, why the program made certain tests rather than others, and indeed how it worked at all. If you take the precaution of sprinkling your programs liberally with explanatory comments, these problems will be minimized.

Comments can be added to any other statement after a colon (statement separator), making the rest of the line a remark. Thus

```
660 LET G = P *453.6
```

has the same effect as

```
660 LET G = P * 453.6 : REM convert Pounds to Grams
```

except that the latter has a remark added to explain what is going on. Do not, therefore, despise the humble REM statement: it can save you many headaches if used wisely.

If execution reaches a REM statement, it is passed over; execution continues with the next statement. So it is quite permissible to have a statement such as GOTO 960 where line 960 is a REM statement.

2.5 Editing

A Basic program is normally built up by typing lines in, each preceded by its line number, on the keyboard. Lines can be entered in any order you like: they are automatically sorted by the Basic system.

Every line must have its own unique number, in the range 1 to 65535. These line numbers serve two purposes: they determine the order in which statements are executed; and they identify lines for correction or insertion when the program is being edited.

It is very likely that when you first type in a program it will contain mistakes. Basic makes it very easy to correct these, for instance by replacing the lines which are incorrect. Consider the effect of typing the following lines.

```
10 REM -- First line
20 REM -- 2nd line
99 END
30 REM -- Third line
20 REM -- New second line
40 REM Line 4
```

The result of typing these lines in this order is to leave the following five lines in the working area of memory, in the order shown.

```
10 REM -- First line
20 REM -- New second line
30 REM -- Third line
40 REM Line 4
99 END
```

The lines have been sorted into ascending sequence, and the second version of line 20 has overwritten the first. If at this point we were to type a line numbered 35 it would be placed between lines 30 and 40.

An incorrect line can be replaced merely by typing its line number and then the line as it should be, ended by RETURN. To delete a line altogether, just type its line number followed immediately by RETURN and that line will be erased from memory.

For deleting large groups of lines the command DELETE is available.

```
DELETE 1-100
```

deletes lines 1 to 100 inclusive (i.e. line 100 and everything preceding it), while

```
DELETE 100-9999
```

deletes lines 100 to 9999 inclusive. (Careful!) The two commands

DELETE 1000–
DELETE –1000

delete respectively all lines from 1000 onwards and all lines up to 1000. These techniques are general to all Basics and are sufficient for most purposes, but the Amstrad microcomputers provide screen-editing facilities to make life much easier.

There is an EDIT command whose format is

EDIT linenumber

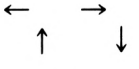
where 'linenumber' specifies the program line you want to amend. Once you give this command, the line concerned is re-displayed and you are able to move along it by using the cursor control keys (← and →). You can enter this editing mode by giving a command such as

EDIT 220

and you will also find that the system puts you into this mode when it halts on account of a syntax error – displaying the offending line as it does so.

All you have to do then to make corrections is to move the cursor left or right with the arrowed keys and perform the deletions and insertions required. Deletion can be done by using DEL in the normal way (deleting characters backwards) or by using CLR (to delete characters forwards). Insertion is accomplished simply by typing text at the point where it is to be inserted. To finish off the line, just press the RETURN key: you do not have to be at the end of the line when you do so.

There is a second method of screen editing as well as the EDIT command described above. The keys with arrows on them at the right hand side of keyboard



enable you to move a flashing cursor around the screen. Hold down SHIFT while you press one of these keys to enter a special editing mode in which the computer displays two cursors. The upper cursor (the 'Read Cursor') is the one you can steer round the screen. The lower one (a small block called the 'Write Cursor') shows what is being entered.

Use the arrowed cursor-control keys (SHIFTed) to position the Read Cursor on whatever you want to copy, then press the COPY key as often as necessary to copy the text, which will appear at the position of the Write Cursor just as if you had typed it in. Parts that are incorrect can be skipped over by the left or right arrow keys and new text inserted by typing in the normal way. When the revised line has been assembled, press RETURN and – Bingo! – the new line is accepted as if it had all been typed in, replacing the

old one. It certainly beats re-typing in full, especially since COPY, like other keys, repeats itself automatically if you hold it down for half a second or so.

Of course the DEL key may be used during editing to erase single characters, leftwards.

Once you get used to the cursor control keys, you will wonder how you ever did without them. They are part of the appeal of the Amstrad computer; but their use is very much a matter of practice, not something that is best learned from a book.

Note: It is advisable to number program lines in steps of five or ten to leave room for new lines to be inserted when the program is being revised. For this the AUTO command is most useful.

AUTO 200,20

provides line numbers 200, 220, 240 and so on without your having to type them. It saves typing and leaves gaps for afterthoughts. To escape from AUTO insertion mode, just press the ESC key.

If the line numbering gets untidy, RENUM can be used for renumbering. For example,

RENUM 100,20,10

renumbers the current program starting at line 20 (which becomes new line 100) and increasing in steps of 10.

2.6 Example program [EASTERS]

In the following example a program to calculate the date of Easter is typed in, then listed and run. The first attempt fails because the program contains errors at lines 170 and 330. It is edited using some of the facilities described in Section 2.5, and the revised program is then re-run and finally SAVEd on disc. (Be warned that the first version contains mistakes. To see the corrected version, look at the second listing.)

Easter Day falls on the first Sunday following the first full moon on or after the Vernal Equinox, 21 March. The following algorithm – reputedly due to the great German mathematician Gauss – will calculate, for any given year (**y%**) between 1582 and 4903 AD, the date of Easter Sunday. (Assuming of course that Easter is still celebrated in 4903 AD.)

The variables used can briefly be described as follows: **cent%** is the century; **greg%** is the 'Gregorian correction', i.e. the number of years divisible by 4 such as 1800 and 1900 when a leap year was not held; **gold%** is the 'golden number' which is used to determine the position of an idealized moon; **c%** is the 'Clavian correction'; and **moon%** is the epact, or age of the moon on 1 January.

The variables are used to compute **d%** in lines 230 to 320 of the program. Then if **d%** is less than or equal to 31, Easter is on March **d%**, otherwise it is on April **d%–31**.

Integer variables (with the trailing % sign) are used here because we are in fact dealing with whole numbers and because integers take less storage space (2 bytes instead of 5) and are handled more quickly than floating-point numbers. Variable names are in lower case throughout.

One of the pleasant features of Amstrad Basic is that keywords are put into CAPITAL letters even if you type them in lower case. Variables may be in upper or lower case, but in this book we normally prefer lower case because that makes the keywords stand out better visually – as landmarks in the program landscape.

Listing 2.1 Easter day calculations

```
[Initial Version]

100 REM ******************************
110 REM ** Listing 2.1 :           **
120 REM ** EASTER DAY CALCULATIONS **
130 REM ******************************
150 PRINT "Program to compute date of Easter Sunday:"
160 PRINT "give date prior to 1582 to end execution.": PRINT
170 INPUY "Which year "; Y%
180 IF Y% < 1582 THEN PRINT "Merry Xmas!";CHR$(7): END
190 IF y% > 4902 THEN PRINT "You should live so long!"
200 REM -- 2nd message is just a warning.
210 REM -- main calculations start here:
220 PRINT "Easter Day falls on ";
230 cent% = Y% \ 100 + 1
240 greg% = 3 * cent% \ 4 - 12
250 gold% = Y% MOD 19 + 1
260 clav% = (8 * cent% + 5) \ 25 - 5 - greg%
270 e% = 5 * Y% \ 4 - greg% - 10
280 moon% = (11 * gold% + 20 + clav%) MOD 30
290 IF moon%=25 AND gold%>11 OR moon%=24 THEN moon%=moon%+1
300 d% = 44 - moon%
310 IF d% < 21 THEN d%=d%+30
320 d% = d% + 7 - (d%+e%) MOD 7
330 IF d%>31 THEN PRINT "April ";d% ELSE PRINT "March ";d%
340 PRINT
350 GOTO 170
360 REM -- goes back for more.
370 That's all folks!

Ready
run
Program to compute date of Easter Sunday
:
give date prior to 1582 to end execution
.

Syntax error in 170
170 INPUt "Which year "; Y%
run
```

```
Program to compute date of Easter Sunday
:
give date prior to 1582 to end execution
.

Which year ? 1982
Easter Day falls on April  42

Which year ? 0
Merry Xmas!
Ready

330 IF d%>31 THEN PRINT "April ";d%-31 ELSE PRINT "March ";d%
```

[Corrected Version]

```
100 REM ******************************
110 REM ** Listing 2.1 :           **
120 REM ** EASTER DAY CALCULATIONS **
130 REM ******************************
150 PRINT "Program to compute date of Easter Sunday:"
160 PRINT "give date prior to 1582 to end execution.": PRINT
170 INPUT "Which year "; Y%
180 IF Y% < 1582 THEN PRINT "Merry Xmas!";CHR$(7): END
190 IF y% > 4902 THEN PRINT "You should live so long!"
200 REM -- 2nd message is just a warning.
210 REM -- main calculations start here:
220 PRINT "Easter Day falls on ";
230 cent% = Y% \ 100 + 1
240 greg% = 3 * cent% \ 4 - 12
250 gold% = Y% MOD 19 + 1
260 clav% = (8 * cent% + 5) \ 25 - 5 - greg%
270 e% = 5 * Y% \ 4 - greg% - 10
280 moon% = (11 * gold% + 20 + clav%) MOD 30
290 IF moon%=25 AND gold%>11 OR moon%=24 THEN moon%=moon%+1
300 d% = 44 - moon%
310 IF d% < 21 THEN d%=d%+30
320 d% = d% + 7 - (d%+e%) MOD 7
330 IF d%>31 THEN PRINT "April ";d%-31 ELSE PRINT "March ";d%
340 PRINT
350 GOTO 170
360 REM -- goes back for more.
370 That´s all folks!

RUN
Program to compute date of Easter Sunday:
give date prior to 1582 to end execution.

Which year?  1982
Easter Day falls on April  11

Which year?  1983
Easter Day falls on April  3

Which year?  1984
Easter Day falls on April  22

Which year?  1985
Easter Day falls on April  7
```

```
Which year?  1986
Easter Day falls on March  30

Which year?  1987
Easter Day falls on April  19

Which year?  1988
Easter Day falls on April  3

Which year?  1999
Easter Day falls on April  4

Which year?  4444
Easter Day falls on April  24

Which year?  5555
You should live so long!
Easter Day falls on April  17

Which year?  1
Merry Xmas!
```

Notice that this program does not run correctly at the first attempt. This is fairly typical, even for a short program. The first RUN threw up an error message

Syntax error at line 170

because INPUT had been spelt INPUY. The correction, using the cursor control and copy keys, was simple. But that was not the end of the story. It turned out that we had put **d%** where we should have put **d%–31** in line 330. (Compare the two versions.) This led the program to the bizarre conclusion that April 42nd was the date of Easter in 1982! Such mistakes (luckily) are easy to spot, since they lead to nonsensical results. But this is not always the case: you should be alert to the possibility that a program giving plausible output may still contain errors.

Having rectified lines 170 and 330, the program finally ran and gave correct answers. Was it worth waiting for, to find out that Easter will be on 4 April 1999?

You may not think so, but this is exactly the sort of fiddly calculation which computers are good at and people (speaking for ourselves) usually get wrong.

Before leaving this example, it is worth mentioning the use of the LIST command to examine portions of the program.

LIST	lists the entire program
LIST n	lists line **n**
LIST n–m	lists lines **n** to **m** inclusive
LIST n–	lists from line **n** onwards
LIST –m	lists up to line **m**

Finally, what about line 370?

It's not Basic but it causes no problem because execution never gets there. The Basic editing system actually allows almost any text to be typed in after a line number, which can be a useful way of data entry.

The reason control never reaches line 370 is that line 350 sends it back to 170. This is a crude form of repetition: we shall see more elegant ways of repeating sections of program in Chapter 3.

The END which eventually halts the program is not at the very end: it is part of the IF statement in line 180. **CHR$(7)** just sounds a buzzer. (More on that in Chapter 6.)

2.7 Quiz

It is all too easy to imagine you are learning when you are not; so here is a short multiple-choice test to enable you to check your understanding of the concepts presented so far.

2.7.1 The questions

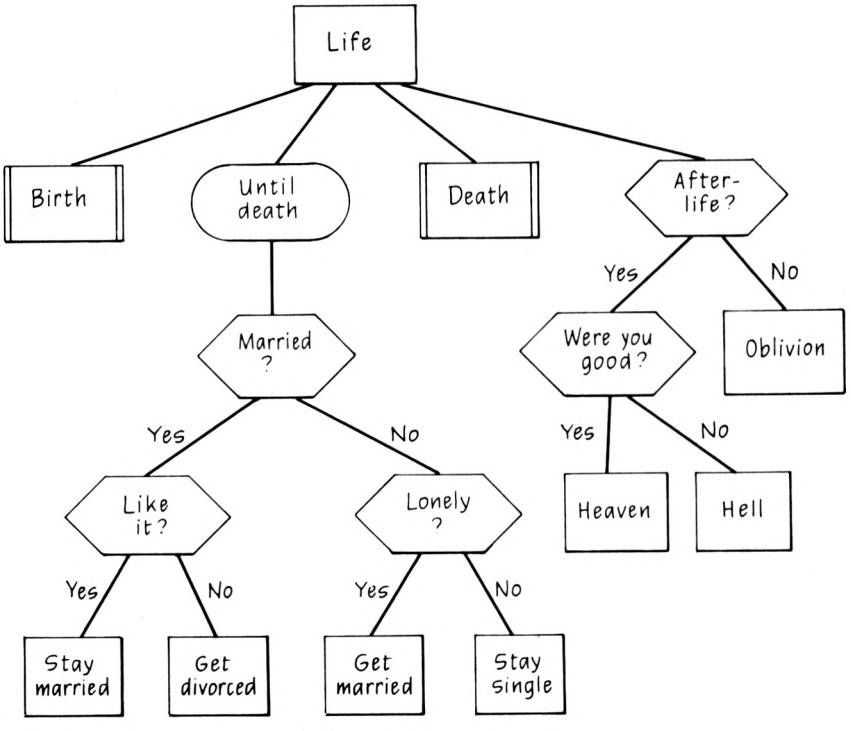
Select one of (a), (b), (c) or (d) as the correct answer and mark it in pencil. The answers are overleaf but do not under any circumstances look at them until you have made a selection for each question.

Q1. A rectangular box with no descendants in a structure diagram signifies
- (a) a simple action?
- (b) a sub-process?
- (c) a decision?
- (d) a repetition loop?

Q2. When a Locomotive Basic program will not stop, it can be forcibly halted by
- (a) typing STOP on the keyboard?
- (b) pressing the BREAK key?
- (c) switching off the power supply?
- (d) pressing ESC?

Q3. Which of the commands below is used to read in a program from disc?
- (a) NEW
- (b) LOAD
- (c) LIST
- (d) OLD

Fig. 2.3 Life, the universe and everything



Q4. The structure diagram above presents a stark view of life. Which of the statements below are true of it? (Only one is.)
(a) Marriage occurs at least once a lifetime.
(b) All good people go to heaven after death.
(c) While there is life there is hope.
(d) Bigamy is not allowed.

Q5. Roughly how many Basic statements would you expect to be required in a program that calculates the day of the week on which Easter falls in any year from 1582 to 4902?
(a) 1
(b) 27
(c) 100
(d) 10

Q6. Only one of the following is a valid Amstrad Basic statement. Which one?
(a) let a + 1 = 0

(b) GOTO 66000
(c) five = 2 + 2
(d) OUTPUT Y

Q7. The value of 10 * 10 − 5 / 2 is
(a) 25
(b) 25.0
(c) 97.5
(d) 47.5

Q8. One of the Basic statements below is invalid. Which one?
(a) REMEMBER THE FIFTH OF NOVEMBER!
(b) LIST 40
(c) LIST 40+80
(d) LIST 40−80

2.7.2 The answers

What do you mean, turning over without even attempting the quiz? You won't learn much just by skimming.

Go on. Do it properly. It won't take long, and it will teach you a thing or two.

A1. (a) Yes.
(b) No. That would have bars down the sides.
(c) No. Decisions have at least 2 descendants.
(d) No. Repetitions are shown by rounded boxes.

A2. (a) Try it: you won't get very far.
(b) No. There is no such key.
(c) Not recommended: too drastic.
(d) Yes, but you may have to press it twice.

A3. (a) No. NEW clears the program area for keyboard entry.
(b) Right.
(c) No. LIST lists the program on the screen.
(d) No. OLD is used to recover after NEW or BREAK.

A4. (a) Not necessarily. Death could occur before marriage.
(b) No. There may be no afterlife.
(c) Nice try, but you are reading between the lines!
(d) Quite right too.

A5. (a) You got it in one: PRINT "Sunday!"
(b) No. You are calculating the date, not the weekday.
(c) Not likely: you really fell for that one.
(d) Sorry, it was a trick question.

A6. (a) No. You need a variable on the left of '=', not an expression.
(b) No: 66000 is too big for a line number (Max. 65535).
(c) Yes, it's an assignment, though it looks rather odd.
(d) No. OUTPUT is wrong; PRINT is the output command.

A7. (a) No. 5/2 ensures a fractional result.
(b) No. The * and / are done before subtraction.
(c) Yes.
(d) No. Read Section 2.2.3 about the precedence of operators.

A8. (a) No. This is a valid REM or REMark statement.
(b) No: this lists program line 40.
(c) Yes: this is wrong.
(d) No: this lists lines 40 to 80 inclusive.

If you have scored less than 5 out of 8 you are already beginning to lose touch. You should re-read Chapters 1 and 2 before continuing, preferably typing in and test-running the example programs to get a feel for the thing.

3

Loops and lists

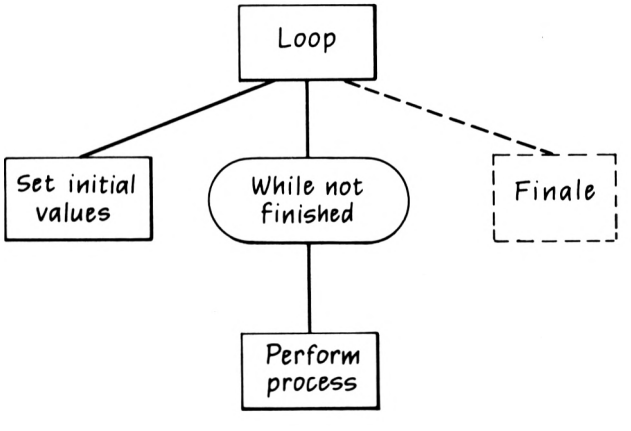
ITERATIVE PROCESSING

One of the concepts that gives the digital computer its great power is the idea of a *loop*. A loop describes a repetitive process; and the computer really comes into its own when required to perform a loop, because although computers are not clever they are very fast.

For that reason one of the arts of computer programming lies in describing a solution in terms of the repetition of a few simple operations over and over again.

In general a loop may be charted as in Fig. 3.1, below. Such a diagram characterizes an iterative process. Notice that a loop has an initialization phase (often forgotten), an exit test, and a process which is repeatedly executed – usually referred to as the 'loop body'. There may also be some sort of 'finale' which tidies up after the loop has run its course.

Fig. 3.1 Generalized loop

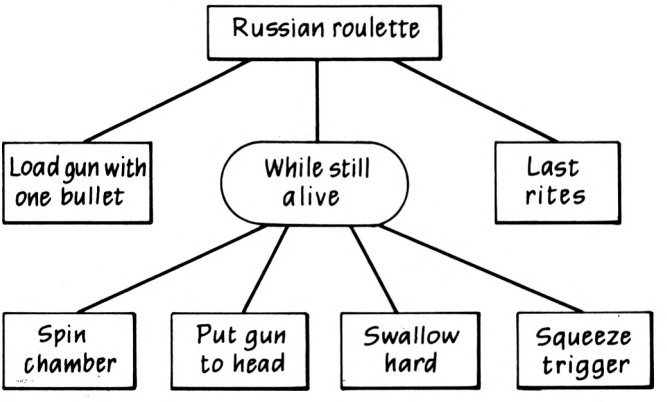


It is important that the actions within the loop body have an effect on some variable used in the exit test so that after a finite number of repetitions the

loop will terminate. Otherwise you have an endless loop, the bugbear of careless programmers.

A more dramatic example is illustrated in Fig. 3.2. This loop has an additional 'finalization phase' after the main process – in which someone has to clear up the mess.

Fig. 3.2 Russian roulette



A loop may be constructed in Basic by using an IF statement for the exit test and a GOTO to cause recycling at the end of the loop body. Older Basics forced this style of programming upon the user. However, looping is such an essential notion that modern implementations, including Amstrad Basic, have special keywords for dealing naturally with repetition.

3.1 Repetition

Amstrad Basic distinguishes two kinds of loop – a deterministic loop that depends on a counter (handled with the FOR/NEXT construction), and an indeterminate loop where the programmer need not know in advance how many repetitions will be carried out (for which the WHILE/WEND pair is provided).

The latter case is more general, but deterministic loops are simpler so we deal with them first.

3.1.1 FOR and NEXT

Below is a little program for printing a Centigrade/Fahrenheit conversion table, so you can see how hot it 'really' is when the thermometer registers 32 degrees Celsius or suchlike. Gasp!

Listing 3.1 Centigrade to Fahrenheit

```
10 REM ******************************
11 REM ** Listing 3.1 :            **
12 REM ** CENTIGRADE TO FAHRENHEIT **
15 REM ******************************
20 REM -- F = 9/5*C + 32
40 PRINT "Celsius","Fahrenheit"
50 FOR c = 0 TO 50
60   f = 9/5 * c + 32
70   PRINT c,f
80   NEXT c
99 END
```

The initialization is carried out in lines 40 and 50; lines 60 and 70 constitute the loop body; line 80 is, in effect, the exit test; and the finale – line 99 – consists simply of ending the program. Try it on your machine.

This illustrates the FOR loop in action. The general form of the FOR statement is

FOR counter = initial TO terminal [STEP stepsize]

where 'counter' is the variable being used as the loop-count, also known as the 'index'. 'Initial' is an arithmetic expression which will be the starting value of the counter, and 'terminal' is another expression placing a limit on the number of repetitions. The 'stepsize', if present, specifies the amount to be added to the index variable each time the loop is repeated. Repetition ceases when that variable passes the terminal value.

If the STEP phrase is omitted (as in the temperature example) an increment of + 1 is assumed. Thus the normal case is to step upwards through all the values from initial to terminal one by one. But other stepsizes, including negative or fractional ones, may be specified. If the stepsize is negative, the counter will be decremented – counting down from initial to terminal.

Each FOR statement must have a matching NEXT. Its form is

NEXT [variable]

where the 'variable' is optional, but if present must be the variable used as the counter by the FOR statement. NEXT closes the loop, whose body consists of all statements enclosed between FOR and NEXT.

In most cases it is advisable to use integers to control FOR loops, both because integers are processed faster and because floating-point stepsizes can lead to round-off errors.

Here is our temperature converter rewritten to take account of this advice. This time a STEP portion is used to make it count downwards in steps of 2 degrees. We have also taken the opportunity to give the variables more meaningful names.

Listing 3.2 Temperature converter

```
10 REM ******************************
11 REM ** Listing 3.2 :            **
12 REM ** TEMPERATURE CONVERTER    **
15 REM ******************************
100 PRINT "Celsius","Fahrenheit"
110 FOR celsius% = 100 TO 0 STEP -2
120   fahr% = 9/5 * celsius% + 32
130   PRINT celsius%,fahr%
140   NEXT
150 END
```

```
Celsius       Fahrenheit
 100           212
 98            208
 96            205
 94            201
 92            198
 90            194
 88            190
 86            187
 84            183
 82            180
 80            176
 78            172
 76            169
 74            165
 72            162
 70            158
 68            154
 66            151
 64            147
 62            144
 60            140
 58            136
 56            133
 54            129
 52            126
 50            122
 48            118
 46            115
 44            111
 42            108
 40            104
 38            100
 36            97
 34            93
 32            90
 30            86
 28            82
 26            79
 24            75
 22            72
 20            68
 18            64
 16            61
 14            57
 12            54
 10            50
 8             46
```

```
6               43
4               39
2               36
0               32
```

Loops may be nested one inside the other, but may not overlap. An inner loop must be totally enclosed within an outer one. Two examples should clarify this rule.

```
100  REM -- Valid Nesting:
120  INPUT rows%,cols%
140  FOR r% = 1 TO rows%
150     FOR c% = 1 to cols%
160        PRINT "*";
170        NEXT c%
175     PRINT
180     NEXT r%
199  END
```

```
100  REM -- Invalid Nesting:
120  INPUT rows%,cols%
140  FOR r% = 1 TO rows%
150  FOR c% = 1 TO cols%
160  PRINT "?";
170  NEXT r%
175  PRINT
180  NEXT c%
199  END
```

Try them. The first will display a rectangle of asterisks (**rows%** deep and **cols%** wide) on the screen. The second should produce a line of question marks followed by an error message.

The error arises in attempting to end the outer loop involving **r%** before the inner one using **c%**. This is not allowed: the loop that starts earlier must finish later.

The first example also presents a style of indentation which exhibits the nesting of loops in visual form. This enhances the legibility of programs and will be used throughout the book. It is a good idea to make such an indentation scheme a habit of your own. Not only does it make programs easier to read, it can also be valuable in spotting mistakes, since a listing that does not return to the left margin indicates a missing NEXT (or WEND)

The second, invalid, example is not indented because no indentation format could be consistent.

3.1.2 WHILE and WEND

Not all loops depend on incrementing or decrementing a counter. The WHILE statement is for loops where the number of repetitions is not known in advance but a terminating condition can be specified. Its form is

WHILE condition

followed by a sequence of statements ending with

WEND

where 'condition' is a Boolean expression that evaluates to true or false as explained in Section 2.4.3.

The statements between WHILE and WEND are executed repeatedly until the condition becomes false. Then execution carries on with whatever comes after the WEND.

Notice the following points.

(1) The exit test is made at the start of the loop, so the loop body may be skipped entirely in some cases.

(2) There may be many statements between an opening WHILE and its closing WEND, including of course embedded WHILE/WEND loops.

Some languages – such as Pascal or BBC Basic – have a REPEAT/UNTIL construction which ensures the loop is always executed at least once. The 'zero-trip loop' is used more than you might think, however, and it is always possible to force a single execution if necessary.

To illustrate this, let us return to our game of Russian roulette, where to avoid the main loop altogether would smack of cowardice!

Listing 3.3 Russian Roulette

```
10 REM ******************************
11 REM ** Listing 3.3 :            **
12 REM ** RUSSIAN ROULETTE         **
15 REM ******************************
100 REM -- Russian Roulette with a 6-gun:
110 maxshots%=6
120 full% = INT(RND*6)+1
130 PRINT "Chamber";full%;"has the bullet."
135 PRINT
140 goes% = 0: bang% = -99
150 WHILE goes% < maxshots% AND bang% <> full%
160   bang% = INT(RND*6)+1
170   PRINT "chamber";bang%;
180   IF bang%=full% THEN PRINT: PRINT "Aarrgh!" ELSE PRINT " click."
190   goes% = goes% + 1
200   WEND
210 IF bang%<>full% THEN PRINT: PRINT "Well done: you survived!"
220 END
```

```
Ready
run
Chamber 1 has the bullet.

chamber 6  click.
chamber 5  click.
chamber 4  click.
chamber 1
Aarrgh!
Ready
run
Chamber 2 has the bullet.

chamber 3  click.
chamber 3  click.
chamber 6  click.
chamber 6  click.
chamber 1  click.
chamber 1  click.

Well done: you survived!
Ready
```

This time the rules are relaxed (slightly!) to permit the possibility of survival. The game ends either with the player's demise or when he has made six spins of the chamber. This is embodied in the dual condition

goes% < maxshots% AND bang% <> full%

linked by the logical connective AND. (N.B. line 120 generates a 'random' integer from 1 to 6 inclusive: more details in Chapter 4.)

A couple of specimen runs are appended. In one our brave Russian survives; in the other he does not.

Try it yourself a few times. How often would you expect a player to get through the complete set of six shots? You can experiment by altering **maxshots%**: the difference between, say, **maxshots%=3** and **maxshots=12** should be dramatic.

3.2 Arrays

In Basic, an array is simply an ordered collection of numbers. (It could also be a collection of strings, but we will leave that till Chapter 5.) The array represents a way of storing a group or aggregation of data under one name. Individual elements or members of the array are picked out by an identifying number known as 'subscript'.

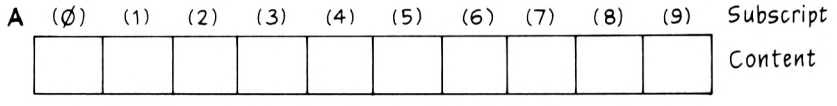
There is a natural relationship between the array and the loop: loops apply repetition to program statements, arrays, on the other hand, apply repetition to data. Consequently, as we shall see arrays are frequently processed by looping.

Arrays have one or more 'dimensions'. A one-dimensional array is termed

a 'vector', or sometimes a 'list'; a two-dimensional array is called a 'matrix' or 'table'. Arrays of three, four and more dimensions are permitted in Amstrad Basic, but we shall not be concerned with them here. Multi-dimensional arrays, as you will find if you use them, eat up storage at a prodigious rate.

A vector is an ordered list of variables. If an ordinary variable is thought of as a storage location for one number then a vector is a row of locations for holding several numbers. A ten-element vector (**A**) is depicted in Fig. 3.3.

Fig. 3.3 Ten-element vector, named **A**

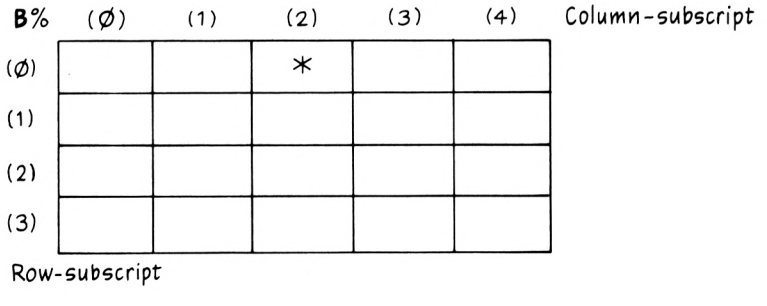


An individual item in this vector is designated by following the array name (**A**) with a numeric expression – the subscript – in parentheses. Thus **A(1)** refers to element **1** and **A(J%)** to element **J%**. Note that the lowest-numbered element is at position zero, not one, so if **J%=7** then **A(J%)** is actually the eighth item in sequence.

A subscript may be a constant, variable or expression. If the subscript has a non-integer value it is rounded to the nearest whole number. Thus **A(5)** and **A(5.25)** both refer to the same element (the sixth item), but **A(5.77)** refers to a different item, namely **A(6)**. If the subscript expression is out of bounds – below 0 or above the array's maximum – Basic reports an execution error and halts the program.

A matrix is essentially a table of values. Matrices are usually described in terms of rows and columns. In order to identify an element of a matrix, two subscripts are required. A twenty-element matrix **B%** with four rows and five columns is illustrated in Fig. 3.4.

Fig. 3.4 Twenty-element matrix, named **B%**



Each box in this diagram is capable of holding one number (in this case an integer), and is denoted by following the array name with two subscripts in parentheses separated by a comma. Conventionally, the first subscript is considered to specify the row and the second the column, so that **B%(0,2)** refers to the cell on row 0, column 2 (not row 2, column 0) – the one containing the asterisk in the figure. Again, the subscripts can be any arithmetic expressions and are rounded if not integral. Subscript values that are out of bounds cause error messages.

An array element may be used anywhere that a variable can be referred to.

Vectors and matrices give the programmer a means of organizing data. In Basic, the array is the most important means of data structuring. Many operations on large collections of data, such as sorting a list into ascending or descending order, would be virtually impossible without arrays.

3.2.1 The DIM statement

The DIM statement (short for DIMension) is used to set up arrays. It allocates a block of storage inside computer memory for an array or arrays. Its form is

DIM name(size) {,name(size)}

and its purpose is to reserve space for one or more arrays. The 'size' will be a single numeric value for a vector or two numeric values separated by a comma in the case of a matrix. Several arrays may be declared in the same DIM statement. Some examples follow.

```
20 DIM DAYS%(12)
22 DIM chessmen%(8,8), Zonk(1000)
23 DIM line%(linemax%)
25 DIM bloc%(255)
```

Line 20 sets up a thirteen-element integer vector called **DAYS%** with subscripts ranging from 0 to 12. Line 22 defines two arrays, a matrix and a vector. The matrix **chessmen%** might well serve to represent the state of play in a chess game. Since subscripts begin with zero it has $9 \times 9 = 81$ elements. This might be thought a wasteful way of holding information on the $8 \times 8 = 64$ squares of a chessboard; but it is sometimes worth sacrificing space for a simpler numbering scheme, and people prefer to count from one rather than nought.

The vector Zonk will contain room for 1001 floating-point numbers. These will occupy 5005 memory-locations. The vector **line%** has its size specified by a variable **linemax%**. This means that its size is determined when the program runs by the contents of **linemax%**, not written into the program. By allowing variable-dimensioned arrays in this way Locomotive Basic permits greater flexibility than many mainframe computer languages.

Finally line 25 allocates a block of 256 contiguous storage locations for integers. This will actually take up 512 bytes of RAM.

Array names follow the same rules as variable names: begin with a letter and continue with letters and/or digits. The percent sign at the end, if present, indicates that the array is for holding integers; if absent it is for floating-point numbers.

The numbers in brackets give the upper limits for the subscripts. These do not have to be constants, as shown on line 23 above. However, once an array has been dimensioned its size is fixed for the rest of the program. It can only be redimensioned by using the ERASE instruction (which erases its contents and reclaims the memory space it used) and then DIMensioning it again from scratch. Thus

```
50 ERASE Zonk
```

would remove all trace of the array **Zonk**. Another array called **Zonk**, with a different shape and size if desired, could be created afterwards.

Amstrad Basic will assume that undeclared arrays have a maximum subscript value of 10. This is meant to be helpful, but in fact it just disguises a number of typing errors that are easily made. You are advised not to rely on this default assumption, but to DIMension all arrays explicitly, even if they have a size of 10.

3.2.2 Matrix operations

Some versions of Basic provide special matrix-handling statements. These can be used for operations on an entire matrix or vector at one time. They are most convenient for 'number crunching' applications on large computers. Amstrad Basic, like most microcomputer implementations, does not support them, so we do not use them in this book.

3.3 Example program [SORT]

Sorting, in computer parlance, means the arrangement of data into ascending or descending sequence.

Computers spend a great deal of their time sorting, mainly in order to make retrieval of information easier, either for man or machine. The reason for this is obvious if you consider how much easier it is in a standard telephone directory (where subscribers are listed alphabetically) to find the number of a particular person than the name belonging to a given phone number.

Our program uses the simple Selection Sort to order a set of numbers. This is one of the least complex (though not the most efficient) ways of sorting algorithms.

It works by making a series of *passes* through a vector. Each pass finds the location of the largest value in the array, and at the end of each pass this maximum value is exchanged with the final element. This puts the last element into the correct place; therefore that element may be ignored on the next pass.

Thus each pass is one step shorter than the previous one; and the process terminates when the next pass would have only one item to consider.

The program makes use of the keyword TIME to measure how long the sorting takes. As you can see, to sort 20 numbers takes more than twice as long as sorting 10 numbers. This is a general feature of all sorting methods.

Selection Sort is not a particularly fast technique. Consequently it is unsuitable for large data sets. It is generally quicker than the well-known, but hopelessly inefficient, Bubble Sort, but markedly inferior to Quicksort or Heapsort.

Listing 3.4 Simple selection sort

```
10 REM ******************************
11 REM ** Listing 3.4 :             **
12 REM ** SIMPLE SELECTION SORT     **
15 REM ******************************
100 REM -- Chapter 3:
110 REM -- Simple Selection Sort,
120 REM -- by R.S. Forsyth, Nov-85.
130 :
140 PRINT "How many items to be sorted ";
150 INPUT n%
160 DIM datlist(n%)
170 FOR i%=1 TO n%
180   INPUT datlist(i%)
190   NEXT
200 :
210 REM -- Now reorder them:
220 t=TIME : REM time at start.
230 last%=n%
240 WHILE last% > 1
250   largest%=last%
260   FOR i%=1 TO last%-1
270     IF datlist(i%) > datlist(largest%) THEN largest%=i%
280     NEXT i%
290   REM -- now swap last with largest:
300   temp=datlist(last%)
310   datlist(last%)=datlist(largest%)
320   datlist(largest%)=temp
330   last%=last%-1
340   WEND
350 t=TIME - t : REM time taken
360 :
370 REM -- now the output:
380 PRINT
390 PRINT "To sort";n%;"numbers took";t/300;" secs."
400 PRINT
410 FOR i%=1 TO n%
420   PRINT i%,datlist(i%)
```

```
430   NEXT
440 PRINT
999 END

How many items to be sorted ? 10
? 77
? 88
? 22
?-4
? 8
? 88
? 888
? 8888
? 0
? 100

To sort 10 numbers took 0.343333333  secs.

 1            -4
 2             0
 3             8
 4             22
 5             77
 6             88
 7             88
 8             100
 9             888
 10            8888


How many items to be sorted ? 20
? 20
? 1948
? 1985
? 1986
? 0
?-99
? 999
? 9999
?-1
? 0
? 23
? 37
? 10
? 1.25
? 31
? 77
? 88
? 99
? 101
? 0

To sort 20 numbers took 1.07666667  secs.

 1            -99
 2            -1
 3             0
 4             0
 5             0
 6             1.25
 7             10
 8             20
 9             23
```

```
10          31
11          37
12          77
13          88
14          99
15          101
16          999
17          1948
18          1985
19          1986
20          9999
```

The sorting is achieved by the WHILE loop on lines 240 to 340. Within that loop is another (a FOR loop) from lines 260 to 280. The inner loop performs one pass through the array **datlist()** from position 1 to position **last%−1**. As a result it sets variable **largest%** to the location of the biggest value in the section of the array being scanned. When the inner loop terminates, lines 300 to 320 swap the last item with the largest. Then 1 is deducted from **last%** (since the final item is now in place) and the process continues until **last%** is less than 2.

Lines 170 to 190 do the input of unordered data. Lines 400 to 440 display the ordered data. The line

```
160 DIM datlist(n%)
```

shows a vector being declared with a variable specifying its size. This means that the array has exactly the required upper limit on each run (10 the first time, 20 the second in our example) so space is not wasted.

Try running the program a few times to see how the time taken rises with the number of items to be sorted. Do not attempt more than 200 items, unless you have plenty of time to kill.

What happens if **n%** is zero? Is the program's behaviour sensible in this extreme case? If not, can you put it right?

3.4 Exercises

Here is some suggested homework to keep you out of mischief for a few hours.

1 Modify the selection sort program of Section 3.3 to give the output in *descending* order, by altering line 410 not line 270. (Hint: use STEP with a negative increment.)

2. The Fibonacci series is the sequence of integers 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 . . . where each term after the first pair is the sum of the previous terms. It was first proposed in the thirteenth century by Leonardo of Pisa as a

(tongue-in-cheek?) model of population growth among breeding rabbits, and has been fascinating mathematicians and biologists ever since.

Write a program to produce this series of numbers, stopping when it has printed a number greater than one million. Although these numbers are integers, you will find that Amstrad's integer variables will not hold numbers of this magnitude, so you will be forced to employ floating-point variables. (Note, however, that no arrays are needed.)

3. Light travels approximately 299 792 km in 1 second. There are 1.609344 km to the mile and 60*60*24**365.2422 seconds in a year. Write a program which reads distances in light-years and prints out the equivalent distances in 'mega-miles'. (By a mega-mile we mean one million miles.) Some practice distances for testing are tabulated below.

	Light-years	Mega-miles
Sun	0.0000157366	92.5
Pluto (average)	0.000684477	4024
Alpha Centauri	4.3	25 277 500
Sirius	8.7	51 142 869
Rigel	500	
Lesser Magellanic Cloud	185 000	
Andromeda Galaxy	2.2 million	

Pretty big eh?

4. Write a program to read in an array of integers and print them out in reverse order. For example, given the input

1 9 4 8 10 2

its output should be

2 10 2 8 4 9 1.

(N.B. This is not a sorting problem.)

5. Write a program to produce a Centigrade/Fahrenheit conversion chart for temperatures from 0°F to 100°C. Use the fact that

C = (F−32) * 5/9.

(See also Section 3.1.1.)

The table should begin as follows.

C	F
−17.7777	0
−17.2222	1
−17	1.4
−16.6667	2
−16.1111	3
−16	3.2
−15.5556	4
−15	5
...	...

In other words, the problem is to coordinate TWO separate series – one stepping through whole numbers of degrees Centigrade, the other through degrees Fahrenheit. This entails a kind of merging. The hard part is to ensure that the two series are properly interleaved. (No arrays are needed in this one either.)

6. An asset depreciates at **P** percent per annum (**0 < P <= 100**) until it reaches its 'salvage value', or rock-bottom resale price.

Write a program to read in an asset's cost, its salvage value and a depreciation rate as a percentage, and to display an annual depreciation schedule. For example, an asset costing £2000 with a salvage value of £1000 would depreciate at 20% as follows.

Year	Depreciation	Value
0	0	2000
1	400	1600
2	720	1280
3	976	1024
4	1000	1000

Make sure that the final year is correctly handled – i.e. that it does not go below the salvage value.

7. Write a program which finds the day in the year given a date in the form DD, MM, YYYY. Thus, given

```
30, 04, 1982
```

for 30th April 1982, it should produce the answer.

```
120
```

indicating that it was the 120th day of 1982.

Leap years should be dealt with correctly: a year that is divisible by 400 is a leap year, so is one divisible by 4 as long as it is not divisible by 100 too. (You can use **MOD** to test for divisibility.)

4

Subprograms

MODULAR PROGRAMMING

A subprogram or routine is any section of a program which performs an identifiable task, and can be called from different parts of the main program to perform that task. Basic provides two kinds of routines – subroutines and functions – though Amstrad Basic does not implement functions in their full generality.

Routines are useful whenever you want to break a long complex job into subtasks, or modules, that can be tested independently, then brought together to make a complete program. Only experience can tell you when and how to use this modular approach, and you will find it rather long-winded when you first try it. You will be handsomely repaid, however, in time saved in program designing, debugging, testing, maintaining and upgrading.

It is not always possible to include every repetition of a given sequence of instructions within a loop, and it is extremely tedious to have to enter the same sequence of instructions at different places in the same program. Even with the COPY facility, it is time wasted. The alternative is to make that sequence a routine in the first place, so that it can be used in different parts of the program simply by 'calling' it, without having to retype the statements that comprise it. Even if a routine is used only once in a program, it can simplify the program in design and appearance by removing detailed code from the main program, leaving it as a 'shell' of easily documented subtasks – writing a title page or a menu on the screen, for example, or reading in a valid date and converting it to a standard format, or sorting a list of numbers.

You can also create a library of useful routines on disc or tape, and select what you need when writing new programs – more time and effort saved. The MERGE command allows pre-written routines to be inserted into the current program (from disc or tape) as if you had typed them at the keyboard.

Routines have two further advantages: they allow you to work with larger program building blocks than the elementary (or 'primitive') operations of Basic; and they allow you to express in a natural way the hierarchical

structure which most non-trivial computations possess. The first facility means that you can customize Basic with your own routines: they are effectively new commands, after all. The second facility simplifies the essential problem of program design into one of problem analysis – if you can describe and then reproduce the structure of the problem, you have almost solved it! The more you use routines and the modular approach to problem-solving, the more effective your programming. Before very long, you will wonder how you ever managed without them.

4.1 GOSUB and RETURN

A routine is any program section that you choose to call a routine. It can be one line or it can be the whole program. That may not be a very helpful statement, but it does illustrate Basic's flexibility in these matters. Of course, a routine is normally a section of code set apart from the main body of the program because it performs an identifiable task, and called into action from more than one point in the program. That is more helpful, but true only by convention: you may find occasions when a routine has none of these characteristics, but is still demonstrably a routine, and clearly useful.

A routine is called by directing control to its first line. GOTO followed by a line number seems the obvious way to do this, but how, when execution is complete, to return control to the point in the program immediately after the routine's call? The whole point of a routine is that it may be called from several different places in the main program. Somehow it must remember where it came from on any given occasion. GOSUB and RETURN provide an answer to this problem. See below.

Listing 4.1 Reversing numbers

```
10 REM ******************************
11 REM ** Listing 4.1 :             **
12 REM ** REVERSING NUMBERS         **
15 REM ******************************
50 N = -99
100 REM -- Numeric reversal and Palindrome tester:
150 WHILE N <> 0
200   t1 = 100000
250   GOSUB 1000 : REM get the input
300   numb = N
350   GOSUB 2000 : REM reverse it
400   PRINT: PRINT "I think you typed",numb;" ?"
450   IF numb=N THEN PRINT "A Palindrome!"
500   GOSUB 2000 : REM reverse it again
550   PRINT: PRINT "Or did you type ",numb;" ?"
600   PRINT
650   WEND
800 END
999 :
```

```
1000 REM -- S/R to get input N:
1010 N = -99
1020 WHILE N<0 OR N>tl
1030   PRINT "Enter a 5-digit number ";
1040   INPUT N
1050   IF N <> INT(N) THEN N=-99 : REM fractions no good.
1060   WEND
1070 RETURN
1080 REM -- result (validated) in N.
1090 :
2000 REM -- Numeric reversal routine:
2010 REM -- uses numb, m, d, k%
2020 m = 0 : REM m is the new version.
2030 FOR k%=1 TO 5
2040   d = numb - INT(numb/10) * 10
2050   m = m * 10 + d
2060   numb = INT(numb/10)
2070   NEXT k%
2080 numb = m : REM inverted value.
2090 RETURN
2100 :

Ready
run
Enter a 5-digit number ? 12345

I think you typed          54321  ?

Or did you type            12345  ?

Enter a 5-digit number ? 12.345
Enter a 5-digit number ? 99999

I think you typed          99999  ?
A Palindrome!

Or did you type            99999  ?

Enter a 5-digit number ? 10201


I think you typed          10201  ?
A Palindrome!

Or did you type            10201  ?

Enter a 5-digit number ? 77

I think you typed          77000  ?

Or did you type            77  ?

Enter a 5-digit number ? 54321

I think you typed          12345  ?

Or did you type            54321  ?

Enter a 5-digit number ? 10000


I think you typed          1  ?

Or did you type            10000  ?
```

```
Enter a 5-digit number ? 0

I think you typed          0  ?
A Palindrome!

Or did you type            0  ?

Ready
```

This demonstrates several interesting points.

(1) The idea has paid off already: the trivial program in Listing 4.1 uses the routine at line 2000 onwards twice, but you have to type those lines only once. If we didn't use a routine you would have to type those lines twice, doubling your chance of making a mistake. Your logical errors in those lines would have to be corrected twice, as well.

(2) We have put the routine after the END so that it cannot be executed in the normal run of the program, but only when control is directed to it.

(3) We have used variables, such as **Numb**, to communicate necessary values to the routine. Such values are termed 'parameters', and the communication itself is called 'passing parameters'. (However, Amstrad Basic does not support parameter passing in the fullest sense, unlike Pascal and certain advanced Basics.)

(4) We have, in a small way, customized Basic. GOSUB 2000 is a new command, and it's all ours!

So routines do make sense, even on this small scale.

When the GOSUB (short for GO to SUBroutine) command is obeyed, Basic automatically notes the position of the next instruction, and when the RETURN is encountered, control passes back to that instruction – even if that instruction is in the middle of a multi-statement line.

Now if we RENUMber the program the line numbers in any GOSUB instructions are adjusted so that they continue to point into their subroutines. The return address is not affected by the RENUM command because it is implied: RETURN tells Basic to go back to just after the last GOSUB which was obeyed.

You can watch the flow and twist of control by issuing the TRON command before you run the program. This causes the interpreter to print the line number [inside square brackets] of each program line as it begins execution. (To cancel this, give the TROFF command.)

It paints a rather confusing picture at first, but if you make the display pause, you will get a chance to study the patterns of program execution. You can make a program pause by pressing ESC (only once) and restart it by pressing any other key (e.g. the space bar). If you press ESC a second time, of course, the program stops altogether, though it can be restarted with the CONT command.

4.1.1 Recursive subroutines

Subroutines can be 'recursive' – they can call themselves, though not without limit, as running this little demonstration will show.

```
100 CLS: deep%=0: GOSUB 200
200 deep%=deep%+1: PRINT "LEVEL "; deep%: GOSUB 200
```

The subroutine increments and prints the value of **deep%**, thus showing the depth of recursion that has been reached, and then calls itself, so descending to the next level – rather like a cat biting its own tail. This could go on forever in theory, but in practice the Basic interpreter runs out of memory for storing return addresses, at which point the systems breaks in with an error message (e.g. "Memory full in 200").

Recursive algorithms are very powerful, but trying to understand them can give you a sore head. It is like peering into parallel mirrors. Consider the following demonstration.

Listing 4.2 The ladder of recursion – *see opposite*

Subroutine 2000 here, which does nothing really beyond demonstrating recursion, is split in half by recursive call

```
2040 GOSUB 2000
```

to itself. The first half of the subroutine is repeatedly obeyed under the influence of line 2040: with each call of the subroutine, the value of **d%** – our depth gauge – increases as the recursion deepens. However, **d%** reaches a limit of 15 and a RETURN is executed from line 2020. This causes control to pass to line 2050 – the next instruction after the GOSUB call at line 2040 – and the second half of the subroutine is repeatedly executed under the influence of the RETURN at line 2080, thereby unwinding the recursion. This goes on until **d%** reaches the top level (not quite back on the surface, but at periscope depth anyway). A final RETURN is carried out in line 2070, sending control back to the main program at line 300.

A simple example like this is not exactly easy to follow, and anything much trickier can become impenetrable – especially when you are trying to debug it. Amstrad Basic makes recursion harder than it ought to be (chiefly because it does not support 'local variables' and named procedures) but it is a fascinating technique nonetheless. Although best avoided in Basic, it is worth knowing about in case you progress to higher (or deeper) things.

```
10 REM ******************************
11 REM ** Listing 4.2 :            **
12 REM ** THE LADDER OF RECURSION  **
15 REM ******************************
100 CLS: d%=0: de%=400
200 GOSUB 2000 : REM dive in!
300 PRINT TAB(0);"Home at last!"
400 END
999 :
2000 REM -- Recursive subroutine:
2010 d%=d%+1: PRINT TAB(d%);"Level";d%
2020 IF d%>=15 THEN RETURN : REM escape route
2030 FOR tick%=1 TO de%: NEXT tick% : REM delay loop
2040 GOSUB 2000
2050 FOR tick%=1 TO de%: NEXT tick% : REM delay
2060 d%=d%-1: PRINT TAB(d%);"LEVEL";d%
2070 IF d%=1 THEN RETURN
2080 RETURN
2090 :
```

```
Level 1
 Level 2
  Level 3
   Level 4
    Level 5
     Level 6
      Level 7
       Level 8
        Level 9
         Level 10
          Level 11
           Level 12
            Level 13
             Level 14
              Level 15
             LEVEL 14
            LEVEL 13
           LEVEL 12
          LEVEL 11
         LEVEL 10
        LEVEL 9
       LEVEL 8
      LEVEL 7
     LEVEL 6
    LEVEL 5
   LEVEL 4
  LEVEL 3
 LEVEL 2
LEVEL 1
Home at last!
```

4.1.2 Pitfalls of using subroutines

Using subroutines can create particular kinds of bugs.

(1) Omitting an END or a RETURN so that control passes to a subroutine by 'falling into' it. This causes an "Unexpected RETURN" error.

(2) Changing main-program variables accidentally by using them inside a routine. This happens most often with loop counters. If you regularly

use **K%** for example, as a loop variable then you might easily write it both in a main program loop and – inadvertently – in a subroutine called within that loop. The value of **K%** would be changed inside the subroutine with unpredictable effects on return to the main program. You might find it worthwhile adopting a convention to avoid this, such as always ending subroutine-only variable names with 9 (**K9**, **LEO9**, **ASI9**, for example).

Point 2 above shows the most important weakness of subroutines: they do not permit *local* variables – i.e. variables that belong only (privately) to the subroutine. Program variables are *global*: they can be accessed anywhere in the program; and according to Sod's Law they probably will be, just at the wrong moment.

In Amstrad Basic there is not much you can do about this, except to take obsessive care about the use of global variables within subroutines.

4.2 Functions and function definition

Functions differ from subroutines in that they can pass parameters (or 'arguments') as part of the call instruction, they have names like variables and they return values like arithmetic expressions. The pre-defined functions (see Appendix F) are part of Basic, and you may have started using some of them, probably without realizing that they were anything out of the ordinary.

4.2.1 Predefined functions

The most commonly used functions are listed below.

Function	Result
ABS(X)	Absolute value of **X**, ignoring sign
COS(X)	Cosine of **X**
EXP(X)	Natural exponent of **X** (**e** to the power of **X**)
FIX(X)	**X** rounded towards zero
INT(X)	Integer part of **X**, those digits to the left of the decimal point (truncated)
LOG(X)	Natural logarithm of **X**, base **e** (**X** > **0**)
RND	Random fraction between zero and one
SIN(X)	Sine of **X**
SQR(X)	Square root of **X**, provided **X** is non-negative
TAN(X)	Tangent of **X**

In the trigonometric functions, such as COS, the numeric argument **X** is normally expressed in radians; but you can convert all trigonometric functions to working with degrees with the DEG instruction.

When you write, for example,

```
100  root = SQR(n)
```

you are calling a subprogram whose name is **SQR**, passing it as its argument the current value of a variable (**n**) and finally treating **SQR(n)** as an expression, just like (**n + 2**) or any other arithmetic expression. So if **n=49**, the value of root would be set to **7**, and likewise with other values of **n**. These pre-defined functions are special cases of the general class of functions.

4.2.2. User-defined functions

You can also create your own functions with the DEF FN (DEFine FunctioN) command and use them in your programs. In effect, you are giving a shorthand name to a long and complicated formula. The format of a function definition is

```
DEF FNvarname [(arguments)] = expression
```

where 'varname' names the function and should follow the normal rules for variable naming. The optional 'arguments' are names for the parameters that will be passed into the function, and the 'expression' on the right computes a value – normally with reference to the arguments on the left. If there are several arguments, they must be separated by commas. Thus

```
50  DEF FNvelocity(rate) = 36 / 16.09344 * rate
```

defines a function called FNvelocity with one argument that converts a speed in metres per second to miles per hour. Given this definition, the statement

```
500  LET speedmph = FNvelocity(ms)
```

is equivalent to

```
500  LET speedmph = 36 / 16.09344 * ms
```

because the expression (**ms**) in the function reference gives its value to the argument rate in the function definition. Here **ms** is a simple variable, but it could be any arithmetic expression.

Although this facility is a rather half-hearted implementation of the concept of a function, it can still be very useful. Notice particularly the following points.

(1) User-defined functions are just like the pre-defined Basic functions, but must be prefixed by FN to indicate to the Basic interpreter that the next

word is the name of a user-defined function. Omitting the FN prefix will cause an "Unknown user function" error.

(2) Unlike subroutines, function definitions should be put somewhere in the program where control will fall into them before they are called. The function is not defined until its DEF statement has been 'executed'.

(3) Functions are not commands, they must always be used as parts of expressions, like variables, so

```
100 SIN(zeta)
```

is illegal, and must be changed to something like

```
100 sine = SIN(zeta)
```

to make a valid Basic statement.

(4) The argument (the variable **zeta** here) of a function definition is a 'dummy' or 'formal' parameter. When the function is called, the actual value of the argument used in the call replaces the formal parameter wherever it occurs in the definition. If there are several arguments, replacement is determined by position, not name: the *n*th expression in the call replaces the *n*th formal parameter in the definition. Argument names are local to the definition in which they appear.

If these concepts seem puzzling, have a look at the following example which prints a value (in the this case Pi) to a varying number of decimal places.

Listing 4.3 Rounding a numeric value

```
10 REM ******************************
11 REM ** Listing 4.3 :           **
12 REM ** ROUNDING A NUMERIC VALUE **
15 REM ******************************
50 :
55 DEF FNroundup(numb,dplaces%)=INT(numb*10^dplaces%+0.5)/10^dplac
es%
60 :
100 PRINT " No. of Dec.  Rounded Value"
110 PRINT " Places       of Pi"
120 FOR n%=0 TO 8
130   PRINT n%,FNroundup(PI,n%)
140   NEXT n%
150 PRINT
160 END
```

```
 No. of Dec.  Rounded Value
 Places       of Pi
 0            3
 1            3.1
 2            3.14
 3            3.142
```

```
4             3.1416
5             3.14159
6             3.141593
7             3.1415927
8             3.14159265
```

The function FNroundup rounds a number up to the stated number of decimal places, given by the second argument. Notice that its arguments are separated by a comma in the definition and the call. Notice too that the pre-defined function INT is employed for truncation within the user-defined function. This is perfectly legal.

Incidentally PI itself is a predefined function, which happens to require no arguments. User functions may also be defined without arguments, in which case the bracketted portion of the definition is omitted entirely. It is also possible to define a function with arguments which are never used. Thus

```
600 DEF FNzero(junk,rubbish)=0
```

is defined with two arguments but does not use them. (We didn't say it was useful!)

4.3 Example program [BISECTOR]

The following program uses a user-defined function and a subroutine, to give you some idea of what such things look like in action. The objective of the whole program is to find the root of an equation (which is defined by a function).

Listing 4.4 Bisector

```
10 REM ******************************
11 REM ** Listing 4.4 :             **
12 REM ** BISECTOR (Example Ch. 4) **
15 REM ******************************
50 :
55 DEF FNroundup(numb,dplaces%)=INT(numb*10^dplaces%+0.5)/10^dplac
es%
60 :
100 CLS: GOSUB 1000 : REM define function of interest.
110 MODE 1
120 PRINT "     -- BISECTOR --"
130 PRINT "  This program finds a root, x0"
140 PRINT "of the equation y = f(x)"
150 PRINT "such that f(x0) = 0 (+ or - a small error)."
160 PRINT "the function f(x) is defined on line 1010"
170 PRINT "in terms of the x-coordinate."
180 PRINT "  The root-finding subroutine on line 2000"
190 PRINT "onwards is called with three parameters:"
200 PRINT "xbtm and xtop enclose a root of f(x)"
210 PRINT "and nearzero is the zero-error."
220 PRINT
```

```
230 PRINT "Hit 1 to GO or 0 to STOP";
240 INPUT n%
250 IF n%=0 THEN STOP
260 xbtm = -30
270 xtop = 30
280 nearzero = 0.002
290 :
300 GOSUB 2000 : REM main routine
310 :
320 IF root% THEN PRINT TAB(7);"ROOT = ";FNroundup(x,4)
330 IF root%=0 THEN PRINT "Root not found after";maxloops%;"bisect
ions."
350 END
999 :
1000 REM -- Routine enclosing function definition:
1010 DEF FNy(x) = (x-1)*(x-6)*(x-12)
1020 RETURN
1030 REM -- alter line 1010 to change equation.
1040 :
2000 REM -- Bisection Subroutine:
2010 REM -- uses xbtm,xtop,x,midvalue,ybtm,ytop,root%,maxloops%
2020 root%=0 : REM root not found.
2030 maxloops%=LOG((xtop-xbtm)/nearzero)/LOG(2)+8
2040 ybtm = FNy(xbtm)
2050 ytop = FNy(xtop)
2055 counter%=0
2060 PRINT "Cycle","Xbtm","Xtop"
2070 WHILE root%=0 AND counter% < maxloops%
2080   x = (xtop+xbtm) * 0.5
2090   midvalue = FNy(x)
2100   IF ABS(midvalue) < nearzero THEN root%=1
2110   IF midvalue*ytop < 0 THEN xbtm=x: ybtm=midvalue
2120   IF midvalue*ybtm < 0 THEN xtop=x: ytop=midvalue
2130   counter%=counter%+1
2140   PRINT counter%,xbtm,xtop
2150   WEND
2160 x = (xtop+xbtm) / 2
2170 RETURN
2180 :

    -- BISECTOR --
  This program finds a root, x0
of the equation y = f(x)
such that f(x0) = 0 (+ or - a small erro
r).
the function f(x) is defined on line 101
0
in terms of the x-coordinate.
  The root-finding subroutine on line 20
00
onwards is called with three parameters:

xbtm and xtop enclose a root of f(x)
and nearzero is the zero-error.

Hit 1 to GO or 0 to STOP? 1

Cycle         Xbtm          Xtop
 1             0             30
 2             0             15
 3             7.5           15
 4             11.25         15
 5             11.25         13.125
```

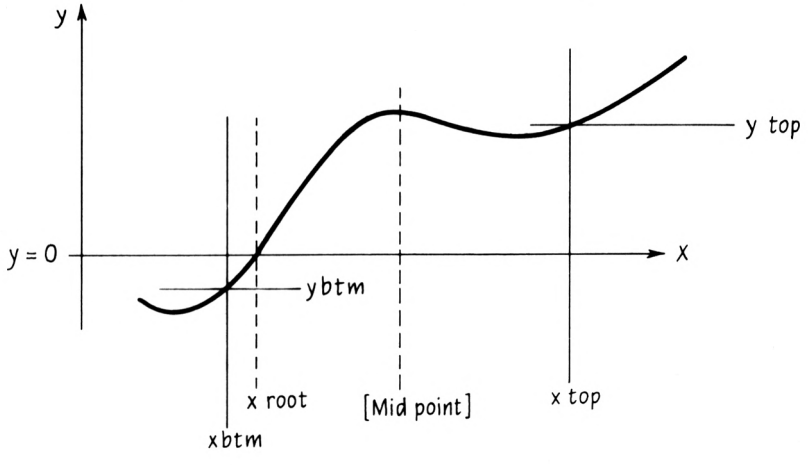
```
 6              11.25          12.1875
 7              11.71875       12.1875
 8              11.953125      12.1875
 9              11.953125      12.0703125
 10             11.953125      12.0117188
 11             11.9824219     12.0117188
 12             11.9970703     12.0117188
 13             11.9970703     12.0043945
 14             11.9970703     12.0007324
 15             11.9989014     12.0007324
 16             11.9998169     12.0007324
 17             11.9998169     12.0002747
 18             11.9998169     12.0000458
 19             11.9999313     12.0000458
 20             11.9999886     12.0000458
      ROOT =    12
Ready
```

This program assumes that you are faced with the problem of knowing an equation, such as

X ∧ 3 − 19*X ∧ 2 + 90*X − 72 = 0

but not knowing how to solve it – that is, how to find values of **X** for which the expression on the left of the equality sign really does equal zero. Here we find a root by simply taking a range of **X**-values which must include a root because the corresponding values of the expression at either end of the range have opposite sign – as in Fig. 4.1.

Fig. 4.1 Range containing a root



We can see that **ybtm** (the value of the expression when **X = xbtm**) is negative, and that **ytop** is positive; somewhere between **xbtm** and **xtop**, therefore, must lie a value of **X** at which the curve crosses the **X**-axis, and at

which the value of the expression is equal to zero. This will be a root of the equation.

We can find values for **xbtm** and **xtop** by inspection or by trial and error: in the expression above, for example, the expression has a large negative value when **X = −30** and a large positive value when **X = 30**. There must be at least one root, then, in this range.

We can find the root by the method of bisection:

(1) Find the midpoint of the range boundaries, and evaluate the function at that point (midvalue);

(2) If midvalue is zero, then the root is found;

(3) If the sign of midvalue is opposite to that of **ytop**, then set **xbtm** equal to the midpoint and repeat from step 1;

(4) If the sign of midvalue is opposite to that of **ybtm** then set **xtop** to midvalue and repeat from step 1.

In the program itself the routine at line 2000 executes this bisection process, given the starting points **xbtm** and **xtop**, and assuming that they are genuine endpoints of a root-enclosing range. The function **FNy** evaluates the expression for the current value of **X**. When the result is smaller in magnitude than a tiny value held in the variable near zero, it is treated as zero.

The routine is called in line 300. Prior to this its parameters (which are in fact global variables) are given the upper and lower range limits.

This subroutine can find roots of a wide range of numeric functions. All you need to alter is the definition of **FNy** on line 1010. (Note that this definition is actually packaged within a subroutine, which is called at line 100 to define the function initially.)

4.4 Exercises

1. Write functions to generate the square, cube and cube root of any number. Generalize them into a pair of subprograms that yield the *n*th power or the *n*th root of a given number.

2. The Drunkard's Walk is a classic simulation, well-suited to home micros:

'The drunk staggers away from his lamppost in the middle of a flat open square; his every step is randomly directed around the compass and there is no method in his steps – but they are always the same length. Simulate his walk, and report constantly on the number of steps taken, and his current distance from the lamppost.'

If you know about co-ordinate geometry, then this is just a programming problem. Do a neat job using functions and subroutines.

If you don't think you know about geometry, then just imagine that the surface of the square is covered by a rectangular grid whose lines run north–south and east–west, and that their central intersection is the lamppost. At any time our man will be standing on or near a grid intersection, and we say that the co-ordinates of that point are its distance east and north of the lamppost. West is the same as negative east, and south is negative north.

We can use the RND function to generate an angle between 0 and 360 degrees, and we can say that his pace length is **L** centimetres: if he takes a step on a compass heading of **H** degrees then **E2** and **N2**, his new co-ordinates, are calculated thus

```
E2 = E1 + L * SIN(H)
N2 = N1 + L * COS(H)
```

where **E1** and **N1** are the co-ordinates of the point that he stepped from to reach (**E2**,**N2**). His distance **D** from the centre is calculated by Pythagoras's Theorem:

```
D ∧ 2 = E2 ∧ 2 + N2 ∧ 2.
```

And that is all the geometry you need. The trigonometric functions – like SIN and COS – are already present in the language, so off you go and solve the problem!

3. Pascal's Triangle looks like this:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
```

After the first row, every element is formed from the sum of the two elements above-left and above-right. Generate the triangle to any depth you think reasonable.

4. Write a program that calculates the time in minutes, hours, days or weeks between one date and another. If you wrote it in good style previously (in answering Exercise 7, Chapter 3), you will find it easy to adapt for this problem. What other time-calculating services can you add on to this program?

5. Prime numbers are important to mathematicians; they are numbers with no factors: no whole number divides into them except one and themselves. Write a program to find prime numbers.

The trick here is realizing – or being told – that the factors of a number come in pairs, so that if you know one, you can calculate the other; one of

every pair of factors must be less than or equal to the square root of the number, so you do not have to search half as exhaustively as perhaps you had thought.

6. A new set of postage stamps is to be issued, as soon as their denominations can be agreed on. There will be four in the set, and you can never put more than six stamps on one envelope. What should the denominations be to ensure good 'coverage' – i.e. ideally, that it should be possible to combine stamps so as to cover all postal charges between one penny and six times the value of the dearest stamp?

7. Perfect numbers are numbers whose factors (including 1) add up to exactly the number itself:

> the factors of 6 are 1, 2, 3;
> the sum of 1, 2 and 3 is 6.

Can you discover the next perfect number, and the one after that?

8. In statistical analysis there are three kinds of 'average' for a group of observations. They are called the mean, the median and the mode.

> The mean is the sum of all the observations divided by the number of observations.
> The median is a single value from the data set, such that, if the observations are put in order, the median is the middle-most.
> The mode is simply the most frequently occurring value.

Thus in the sample (12, 14, 14, 14, 23, 25, 29, 33, 39), the mean is 22.5; the median is 23, and the mode is 14.

Write a program to find these values for any collection of observations.

5

Character strings

NON-NUMERIC COMPUTING

A character string is any sequence of characters that is to be treated by Basic as non-numeric. Thus,

23456.789	is a numeric sequence,
"23456.789"	is non-numeric.

The distinction is not immediately obvious. If we run these two lines:

```
100  PRINT 23456.789
200  PRINT "23456.789"
RUN
 23456.789
23456.789
```

there seems no difference in the output; but if we add these two lines:

```
300  PRINT 23456.789 * 4
400  PRINT "23456.789" * 4
```

then Basic executes the first three, but stops at the fourth line with a "Type mismatch" error message. Evidently there is a difference and it can only be the fact that the number is in quotes in one case and not in the other. The quotes, in fact, label their contents as non-numeric, which is why line 400 causes a type mismatch: "23456.789" is the wrong type of data to include in an arithmetic expression, whereas 23456.789 (without the quotes) is obviously a number, and, therefore, entirely suited to arithmetic expressions.

Strings, then, are non-numeric data that we cannot include in arithmetic expressions; so what can we do with them, and what are they for?

The answers are as bald as the questions. We can concatenate them, as we can slice them, and we can compare them; and they are primarily for text processing. This last is their true importance, of course, and it has been noticeably lacking so far from our catalogue of Amstrad Basic facilities. To set this aright, then, we need to know how to input text, how to store it, how to manipulate it, and how to print it out.

5.1 String variables

String variables follow exactly the same meaning conventions as numeric variables with the addition of a dollar (**$**) character at the end of the name. Thus

word$ pronounced 'word-string' or 'word-dollar'
L2$ pronounced 'L2-string' or 'L2-dollar'
million$ pronounced 'million-string' or 'million-dollar'
Shoe$ pronounced 'Shoe-string' or 'Shoe-dollar'

are all legal string variable names.

Like numeric variables, string variables can be assigned values in the program or can be the object of the INPUT statement, as below.

```
100  welcome$ = "Hello"
200  INPUT "WHAT'S YOUR NAME"; name$
300  PRINT welcome$ name$
RUN
WHAT'S YOUR NAME? eric
Helloeric
```

In this program the string variable **welcome$** is assigned the value "Hello", while **name$** takes its value from the keyboard in the course of the INPUT statement. In line 300 the two variables occur one after the other in the PRINT statement, and the effect of this is that the contents of **name$** – whatever we entered in response to the INPUT command – are printed directly after the contents of **welcome$**; a semicolon is not necessary here because the dollar signs tell Basic where one variable ends and the other begins.

You will notice that the program prints "Helloeric". There is no space between the two words, even though one was typed as the first character of the input. If you change line 100, however,

```
100  welcome$ = "Hello "
```

putting a space after the 'o' of 'Hello', then when you run the program again that space will separate the two words in the output, as follows.

```
Hello eric
```

This shows that string variables can contain trailing spaces (spaces at the end of the string), while the INPUT instruction ignores leading spaces (spaces at the start of the string). In fact string variables can contain both leading and trailing spaces.

By experimenting further with line 100 and what you enter when you run the program, you will find that string variables can contain any combination of the characters you can type (and some that you cannot, but more on that

later), though there seems no way to include the quotation symbol (") as part of a string. Nor does it seem possible to enter a comma as part of your response to the INPUT statement. We return to the first point later, and the second point follows from the way that INPUT works: the comma signals the end of an input item, so that several data items can be entered consecutively. An amended version of the program demonstrates this.

```
100  welcome$ = " Hello "
200  INPUT "Enter first-name, second-name "; forename$,surname$
300  PRINT welcome$ surname$ forename$
RUN
Enter first-name, second name ? Grace,Darling
 Hello DarlingGrace
```

In the input, the comma separates the first name ("Grace", stored in **forename$**) from the second name ("Darling", stored in **surname$**). Without this convention the INPUT statement could accept only one data item per line, since there would be no way of telling where one stopped and another started. The comma is called a 'delimiter' when it is used in this way. The character generated when you press RETURN is called a 'terminator', but also acts as a delimiter. Suppose we enter only one name to the program above, and then press RETURN:

```
RUN
Enter first-name, second-name ? Grace
?Redo from start
Enter first-name, second-name ? Grace,Darling
 Hello DarlingGrace
```

The RETURN delimits the first name and moves the cursor to the line below, but the INPUT instruction displays an error message to show that it has not received enough data. Both names then have to be re-entered.

Now make these changes in the program to demonstrate how strings can be concatenated (literally 'chained together')

```
300  greetings$ = welcome$ + surname$ + " " + forename$
400  PRINT greeting$
RUN
Enter first-name, second-name ? Ivan,Terrible
Hello Terrible Ivan
```

The assignment statement in line 300 has joined four separate string values into one, **greeting$**. The plus sign is not performing the same operation as in a numerical expression, precisely because Basic recognizes the type of data involved and performs the appropriate manipulation – concatenation of strings, addition of numeric quantities. It would be handy if a minus sign in a string expression had a corresponding effect, but adding something like

```
500  greeting$ = greeting$ - surname$
600  PRINT greeting$
```

will simply cause a "Type mismatch" error at line 500. You can concatenate strings using the plus sign, but you cannot split them up with the minus sign.

In the line

```
greeting$ = welcome$ + surname$ + " " forename$
```

the item " " is a string constant, or 'literal'. It contains a single space within quotation marks, and was put in to prevent the names running together in the output. We could easily have used another literal, as in

```
greeting$ = " Hello " + surname$ + " " + forename$
```

which would have exactly the same effect. The point of using a variable, **welcome$**, is to achieve flexibility. If we later change line 100 to

```
100  welcome$ = " Hi there "
```

we get a new style of greeting without altering line 300 (and any other lines where **welcome$** might be used). Literals cannot be changed during program execution, whereas the contents of string variables are infinitely variable: that is what they are there for!

One use for string concatenation is to find the maximum permitted length – meaning the maximum number of characters – of a string, as below.

```
100  a$=""
200  FOR k% = 0 to 9999
300     PRINT k%
400     a$=a$+"*"
500     PRINT a$
600     NEXT k%
700  END
```

This program increases the number of characters stored in **a$** one by one until the system limit is reached, and execution stops with a "String too long" error. Notice that in line 100 **a$** is given the value "" known as the 'null', or empty, string. This contains no characters, and has zero length. It is the string equivalent of numeric zero. String variables are automatically set to the null string by the RUN command, so line 100 is not strictly necessary here. It would be necessary, though, if **a$** had been used earlier in the program, and might thus contain some spurious characters.

In Amstrad Basic the maximum string length is 255 characters.

Everything that has been said here about string variables is true for string arrays, which are created by the DIM statement, just like numeric arrays (see Section 3.2.1). We could rewrite our little greetings program to employ a two-dimensional array, as in Listing 5.1.

Listing 5.1 String input

```
10 REM ******************************
11 REM ** Listing 5.1 :            **
12 REM ** STRING INPUT             **
15 REM ******************************
60 :
100 REM -- Greeting a Group:
110 welcome$ = " Hello "
120 INPUT "How many of you are there "; howmany%
130 DIM name$(howmany%,2)
140 FOR n% = 1 TO howmany%
150   PRINT n%;"Enter forename,surname ";
160   INPUT name$(n%,1), name$(n%,2)
170   NEXT n%
180 PRINT
190 :
200 FOR n%=1 TO howmany%
210   greeting$ = welcome$ + name$(n%,2) + name$(n%,1)
220   PRINT n%, greeting$
230   NEXT n%
250 END
```

```
                          How many of you are there ? 4
 1 Enter forename,surname ?Grace,Darling
 2 Enter forename,surname ?Peter,Pan
 3 Enter forename,surname ?Boy,George
 4 Enter forename,surname ?Catherine,the Great
 1            Hello DarlingGrace
 2            Hello PanPeter
 3            Hello GeorgeBoy
 4            Hello the GreatCatherine
```

Here we have stored a group of names in the array **name$**, with the forename in the first column and the surname in the second. Each row is used for a different person.

5.2 String functions

To understand the use of strings you really need to know how Basic stores them. Internally they are held as numbers: that is the only kind of data the computer can store. So every printable character is allocated a code number, called its ASCII code, according to the American Standard Code for Information Interchange (see Appendix A) Two predefined functions, **ASC()** and **CHR$()**, refer specifically to these codes.

ASC(x$) returns the ASCII code of the first character of the string argument **(x$)**.

CHR$(x) returns the character whose ASCII code is the number given as argument **(x)**.

The following program uses the **ASC()** function to display the code number

of any character you type. (It also uses **INKEY$** for single character input, but we will come to that in Chapter 6.)

Listing 5.2 ASCII values

```
10 REM ******************************
11 REM ** Listing 5.2 :            **
12 REM ** ASCII VALUES             **
15 REM ******************************
60 :
100 REM -- Loop till ASCII 0
110 char$ = ""
120 WHILE char$ <> CHR$(0)
130   char$=INKEY$
140   IF char$="" THEN GOTO 130
144   char% = ASC(char$)
150   IF char%<>0 THEN PRINT char%;TAB(8);"is the code for ";char$
160   WEND
200 END
```

```
 32     is the code for
 9      is the code for
 33     is the code for !
 34     is the code for "
 35     is the code for #
 36     is the code for $
 37     is the code for %
 38     is the code for &
 39     is the code for '
 40     is the code for (
 41     is the code for )
 81     is the code for Q
 119    is the code for w
 101    is the code for e
 114    is the code for r
 116    is the code for t
 121    is the code for y
 117    is the code for u
 105    is the code for i
 111    is the code for o
 112    is the code for p
 64     is the code for @
```

And this one prints the entire printable character set:

Listing 5.3 The ASCII character set

```
10 REM ******************************
11 REM ** Listing 5.3 :            **
12 REM ** THE ASCII CHARACTER SET  **
15 REM ******************************
60 :
100 REM -- Tabulates ASCII characters:
110 FOR code% = 32 TO 127
120   PRINT code%;"-> ";CHR$(code%),
140   NEXT code%
150 PRINT
160 END
```

```
32 ->        33 -> !      34 -> "      35 -> #
36 -> $      37 -> %      38 -> &      39 -> '
40 -> (      41 -> )      42 -> *      43 -> +
44 -> ,      45 -> -      46 -> .      47 -> /
48 -> 0      49 -> 1      50 -> 2      51 -> 3
52 -> 4      53 -> 5      54 -> 6      55 -> 7
56 -> 8      57 -> 9      58 -> :      59 -> ;
60 -> <      61 -> =      62 -> >      63 -> ?
64 -> @      65 -> A      66 -> B      67 -> C
68 -> D      69 -> E      70 -> F      71 -> G
72 -> H      73 -> I      74 -> J      75 -> K
76 -> L      77 -> M      78 -> N      79 -> O
80 -> P      81 -> Q      82 -> R      83 -> S
84 -> T      85 -> U      86 -> V      87 -> W
88 -> X      89 -> Y      90 -> Z      91 -> [
92 -> \      93 -> ]      94 -> ^      95 -> _
96 -> `      97 -> a      98 -> b      99 -> c
100 -> d     101 -> e     102 -> f     103 -> g
104 -> h     105 -> i     106 -> j     107 -> k
108 -> l     109 -> m     110 -> n     111 -> o
112 -> p     113 -> q     114 -> r     115 -> s
116 -> t     117 -> u     118 -> v     119 -> w
120 -> x     121 -> y     122 -> z     123 -> {
124 -> |     125 -> }     126 -> ~     127 ->
```

The last program starts listing character codes from number 32. The reason for this is that codes under 32 have special effects. Appendix A gives details of all these 'control characters'. Printing them is a convenient way of sending formatting information to the printer and/or screen. It is a method that springs from ASCII's original definition as a teletype code: the codes from 1 to 31 were generated by control keys on the teletype's keyboard.

One particular use of **CHR$()** is for manipulating such unprintable control characters, especially " and RETURN. The following example should be self-explanatory.

```
100  qq$ = chr$(34): cr$ = chr$(13)
200  line$ = "Insert quotes " + qq$ + cr$ + "and RETURN."
300  PRINT line$
RUN
Insert quotes "
and RETURN.
```

Because strings are stored as numbers, they can be compared using the numeric comparison operators (=, <, >, <>, <=, >=). Thus, one character is less than another if it has a lower ASCII code. When multi-character strings are compared, their first characters are the basis of comparison; if they are equal then the second characters are compared, and so on until the last character of one or both strings is reached. If they both end together, with no difference found, they must be equal; if not, the shorter one comes first (i.e. is lesser). This can lead to some odd results, as shown in listing 5.4.

Listing 5.4 Comparing strings

```
10 REM ******************************
11 REM ** Listing 5.4 :            **
12 REM ** COMPARING STRINGS        **
15 REM ******************************
60 :
100 REM -- Compares character sequences:
110 string1$="*": string2$="!!"
120 WHILE string1$<>"" OR string2$<>""
125   PRINT
130   INPUT "Enter first string :",string1$
140   INPUT "Enter second string:",string2$
150   IF string1$=string2$ THEN PRINT string1$;" is identical to
 ";string2$
160   IF string1$>string2$ THEN PRINT string1$;" is greater than
 ";string2$
170   IF string1$<string2$ THEN PRINT string1$;" is smaller than
 ";string2$
180   WEND
190 PRINT
200 END

Enter first string :aaaa
Enter second string:bb
aaaa is smaller than bb

Enter first string :aaaa
Enter second string:BB
aaaa is greater than BB

Enter first string :CAIN
Enter second string:able
CAIN is smaller than able

Enter first string :Cain
Enter second string:Abel
Cain is greater than Abel

Enter first string :99
Enter second string:1001
99 is greater than 1001

Enter first string :wxyz
Enter second string:abcd
wxyz is greater than abcd

Enter first string :stringy
Enter second string:stringy
stringy is smaller than stringy

Enter first string :Zonk
Enter second string:Bonk
Zonk is greater than Bonk

Enter first string :good
Enter second string:BYE
good is greater than BYE

Enter first string :
Enter second string:
 is identical to
```

In the 'stringy' case, for example, the second string has a trailing space, and so is greater than the 'same' string, without a trailing space. When you are comparing strings, be sure to strip off leading and trailing spaces if you want to get anything like 'lexicographic' order (as in a dictionary). Even then, the way that the computer stores upper and lower case letters and the ASCII values for punctuation marks like space and hyphen can lead to some peculiar orderings.

Strings can be sliced up by using the pre-defined functions **LEFT$()**, **RIGHT$()** and **MID$()**.

LEFT$(a$,n)	returns the **n** leftmost characters of **a$**.
RIGHT$(a$,n)	returns the **n** rightmost characters of **a$**.
MID$(a$,p,n)	returns the **n** characters from position p in the string **a$** onwards.

These functions are frequently used in conjunction with another pre-defined function **LEN()** which returns the length of its string argument – i.e. how many characters it contains.

Listing 5.5 demonstrates these four functions in action.

Listing 5.5 String functions

```
10 REM ******************************
11 REM ** Listing 5.5 :            **
12 REM ** STRING FUNCTIONS         **
15 REM ******************************
60 :
100 REM -- Puts built-in string functions through their paces:
110 PRINT "Enter any string : ";
120 INPUT "", text$
130 :
140 size%=LEN(text$)
200 PRINT "---- LEFT$ ----"
300 FOR s%=1 TO size%
400   PRINT LEFT$(text$,s%)
500   NEXT s%
600 :
700 PRINT "---- RIGHT$-----"
800 FOR s%= 1 TO size%
900   PRINT RIGHT$(text$,s%)
1000  NEXT s%
1100 :
1200 PRINT "----MID$----"
1300 FOR s%=1 TO size%
1400   PRINT MID$(text$,s%,2)
1500   NEXT s%
1600 :
1700 PRINT "----CHUNKS----"
1800 FOR s%=1 TO size%
1900   newtext$=LEFT$(text$,s%)+RIGHT$(text$,s%)
2000   PRINT newtext$
2100   NEXT
2200 :
2300 PRINT
2400 END
```

```
Enter any string : ALPHABET
---- LEFT$ ----
A
AL
ALP
ALPH
ALPHA
ALPHAB
ALPHABE
ALPHABET
---- RIGHT$-----
T
ET
BET
ABET
HABET
PHABET
LPHABET
ALPHABET
----MID$----
AL
LP
PH
HA
AB
BE
ET
T
----CHUNKS----
AT
ALET
ALPBET
ALPHABET
ALPHAHABET
ALPHABPHABET
ALPHABELPHABET
ALPHABETALPHABET
```

Now, with the addition to our string-handling repertoire of the function

INSTR(a$,b$) returns the starting position of the substring **b$** in the main string **a$**, or 0 if **b$** is not found in **a$**.

we can rewrite Listing 5.1 so that first and second names can be entered in a more natural manner, with a space between them. The program reads a name, splits it into its two components (**forename$** and **surname$**), reverses their order and puts the whole lot into upper case.

Listing 5.6 Welcome back

```
10 REM ******************************
11 REM ** Listing 5.6 :            **
12 REM ** WELCOME BACK             **
15 REM ******************************
60 :
100 REM -- First find a gap:
110 welcome$ = " Hello "
120 spos%=0
130 WHILE spos% = 0
```

```
140   INPUT "Enter Firstname Secondname ", name$
150   spos% = INSTR(name$," ")
160   REM -- finds a blank.
170   WEND
180 :
190 size%=LEN(name$): newname$=""
200 REM -- Now capitalize newname$:
210 FOR n%=1 TO size%
220   ch$=MID$(name$,n%,1)
230   IF ch$>="a" AND ch$<="z" THEN ch$=CHR$(ASC(ch$)-32)
240   newname$ = newname$ + ch$
250   NEXT n%
260 :
270 REM -- Now the output:
280 forename$=LEFT$(newname$,spos%)
290 surname$=RIGHT$(newname$,size%-spos%)
300 PRINT welcome$ surname$ ", " forename$
310 GOTO 120
320 REM -- carries on till ESCape.
330 END
```

```
Enter Firstname Secondname HIPPY HAPPY
 Hello HAPPY, HIPPY
Enter Firstname Secondname Washington Irving
 Hello IRVING, WASHINGTON
Enter Firstname Secondname Flamenco Pretty
 Hello PRETTY, FLAMENCO
Enter Firstname Secondname Sir Clive Sinclair
 Hello CLIVE SINCLAIR, SIR
Enter Firstname Secondname John Bull Esq.
 Hello BULL ESQ., JOHN
Enter Firstname Secondname Pretty Flamingo
 Hello FLAMINGO, PRETTY
Enter Firstname Secondname Go Away
 Hello AWAY, GO
```

Notice that **newname$** is assigned the null string in line 190. Try deleting this statement and then run the program. (When you build a string in a loop, always make sure you know what it contains at the beginning of the loop!)

Finally, there are some additional string functions that we have not so far mentioned.

LOWER$(x$)	returns a string the same as **x$**, but with all alphabetic characters converted to lower case.
STRING$(n,x$)	returns a string comprising **n** copies of **x$**.
STR$(n)	returns a string representation of the numeric value **n**.
UPPER$(x$)	returns **x$** with all letters converted into upper case (CAPITAL letters).
VAL(x$)	returns the numerical value of **x$**, provided **x$** represents a number; returns 0 if the first character in **x$** is non-numeric.

STR$() and **VAL()** are inverses. **STR$()** takes an ordinary numeric value and translates it into a string (as would happen if that number were printed out). Thus **STR$(37.25)** would give the result " **37.25**". **VAL()** does the opposite. Thus **VAL("−1985")** would yield the numeric value **−1985**. As we saw earlier, arithmetic operations cannot be applied to strings – even if they look

like numbers. Sometimes it is useful to be able to get round this restriction. **STR$()** and **VAL()** allow you to interconvert freely between numbers and digit-strings.

UPPER$() and **LOWER$()** are also inverse functions, in a sense. **UPPER$()** places any letters in the string argument into capitals, while **LOWER$()** puts any capital letters into lower case. In Basic **"Richard"** is not the same string as **"RICHARD"**, and this distinction can sometimes lead to misleading results – for example if your program is searching for **"CAPS"** it will fail to find **"Caps"**. By using one or other of these functions (consistently!) to put all string searches into the same case, your programs will perform word-matching successfully given any mixture of upper and lower case.

5.3 Example program [NUMBERS]

To demonstrate the usefulness of these various string-handling functions, our example program involves both strings and numbers. The program translates a number given as input into the words, in English, which name that number. For instance 7708 becomes SEVENTY-SEVEN THOUSAND AND EIGHT.

It is the sort of facility you might want in a cheque-printing program, and it shows that Basic is actually quite a powerful string-handling language.

Listing 5.7 Number naming

```
10 REM ******************************
11 REM ** Listing 5.7 :            **
12 REM ** NUMBER NAMING            **
15 REM ******************************
60 :
100 REM -- Puts Figures into Words:
110 GOSUB 2000 : REM initialize number names etc.
120 continue%=1
125 REM -- Main loop:
130 WHILE continue%
140   ok%=0
150   WHILE ok% = 0
160     REM -- Input a number:
170     GOSUB 3000 : REM sets ok%, numb
180     WEND
190   REM -- Standardize numb:
200   GOSUB 4000
210   REM -- Print it in 3 bites:
220   FOR bite% = 1 TO bitenum%
230     GOSUB 5000 : REM print one chunk.
240     NEXT bite%
250   WEND
260 :
270 PRINT "Bye!";CHR$(7)
300 END
```

```
999 :
1000 REM -- Numeral naming data:
1010 DATA "",ONE ,TEN ,""
1020 DATA THOUSAND,TWO ,ELEVEN,"TWENTY "
1030 DATA MILLION,THREE ,TWELVE ,"THIRTY "
1040 DATA BILLION,FOUR ,THIRTEEN ,"FORTY "
1050 DATA FIVE , FOURTEEN ,"FIFTY "
1060 DATA SIX ,FIFTEEN ,"SIXTY "
1070 DATA SEVEN ,SIXTEEN ,"SEVENTY "
1080 DATA EIGHT ,SEVENTEEN ,"EIGHTY "
1090 DATA NINE ,EIGHTEEN ,"NINETY "
1100 DATA "NINETEEN "
1110 :
2000 REM -- Initialization Routine:
2010 DIM bite$(5),zero$(2)
2020 DIM s$(2), bval$(5),unit$(9),teen$(10),tval$(9)
2030 FOR n%=1 TO 9
2040   IF n%<5 THEN READ bval$(n%)
2050   READ unit$(n%),teen$(n%),tval$(n%)
2060   NEXT n%
2070 READ teen$(10)
2080 s$(0)="MINUS": s$(1)="ZERO"
2090 s$(2)="": zero$(0)=""
2100 zero$(1)="00": zero$(2)="0"
2120 RETURN
2200 :
3000 REM -- Input routine:
3010 ok%=0
3015 PRINT
3020 PRINT "Enter an integer"
3030 PRINT "up to 10 digits long ";
3040 INPUT numb$
3050 n=ABS(VAL(numb$))
3060 IF n>1E+10 THEN ok%=0: RETURN
3070 IF n=INT(n) THEN ok%=1
3080 numb = VAL(numb$)
3100 RETURN
3110 :
4000 REM -- Standardization routine:
4010 IF numb <= 999999999 THEN numb$=STR$(ABS(numb)) ELSE numb$=
" "+numb$
4020 size%=LEN(numb$)-1
4025 numb$=RIGHT$(numb$,size%)
4030 sign$=s$(SGN(numb)+1)
4040 IF sign$="ZERO" THEN PRINT sign$: RETURN
4050 numb$=zero$(size% MOD 3)+numb$
4060 size%=LEN(numb$)
4070 FOR p%=1 TO size% STEP 3
4080   bitenum% = INT(p%/3)+1
4090   bite$(bitenum%)=MID$(numb$,p%,3)
4100   NEXT p%
4110 IF sign$="MINUS" THEN numb$="-"+numb$: continue%=0
4120 PRINT numb$ : REM for confirmation.
4130 RETURN
4140 :
5000 REM -- Word-printing routine:
5010 word$=bite$(bite%)
5015 IF sign$="ZERO" THEN RETURN
5020 IF word$="000" THEN RETURN
5030 hundreds%=VAL(LEFT$(word$,1))
5040 tens%=VAL(MID$(word$,2,1))
5050 ones%=VAL(RIGHT$(word$,1))
5060 word$="" : REM clear for re-building.
```

```
5070 IF hundreds%<>0 THEN word$=unit$(hundreds%)+" HUNDRED": IF
tens%+ones%<>0 THEN word$=word$+" AND "
5080 IF tens%=1 THEN word$=word$+teen$(ones%+1) ELSE IF tens%<>0
 THEN word$=word$+tval$(tens%)
5090 IF ones%<>0 AND tens%<>1 THEN word$=word$+unit$(ones%)
5100 IF hundreds%=0 AND bite%=bitenum% AND bitenum%<>1 THEN word
$="AND "+word$
5120 word$=word$+" "+bval$(bitenum%+1-bite%)
5150 IF bite%=1 THEN PRINT sign$
5160 PRINT word$
5180 RETURN
5190 :

  Ready
  run

  Enter an integer
  up to 10 digits long ? 1
  001

  ONE

  Enter an integer
  up to 10 digits long ? 1234
  001234

  ONE THOUSAND
  TWO HUNDRED AND THIRTY FOUR

  Enter an integer
  up to 10 digits long ? 12.34

  Enter an integer
  up to 10 digits long ? 12345678
  012345678

  TWELVE MILLION
  THREE HUNDRED AND FORTY FIVE THOUSAND
  SIX HUNDRED AND SEVENTY EIGHT

  Enter an integer
  up to 10 digits long ? 123456789
  123456789

  ONE HUNDRED AND TWENTY THREE MILLION
  FOUR HUNDRED AND FIFTY SIX THOUSAND
  SEVEN HUNDRED AND EIGHTY NINE

  Enter an integer
  up to 10 digits long ? 1234567890
  001234567890

  ONE BILLION
  TWO HUNDRED AND THIRTY FOUR MILLION
  FIVE HUNDRED AND SIXTY SEVEN THOUSAND
  EIGHT HUNDRED AND NINETY

  Enter an integer
  up to 10 digits long ? 12345678901

  Enter an integer
  up to 10 digits long ? 7007
  007007
```

```
SEVEN THOUSAND
AND SEVEN

Enter an integer
up to 10 digits long ? 1986
001986

ONE THOUSAND
NINE HUNDRED AND EIGHTY SIX

Enter an integer
up to 10 digits long ? 90000099
090000099

NINETY  MILLION
AND NINETY NINE

Enter an integer
up to 10 digits long ? 90909
090909

NINETY  THOUSAND
NINE HUNDRED AND NINE

Enter an integer
up to 10 digits long ? -88
-088
MINUS
EIGHTY EIGHT
Bye!
Ready
```

The program solves what seems to be a somewhat slippery problem by a reductionist method – useful for problem-solving in general. When you are faced with a large difficult problem which you cannot quite get your mind around, try a smaller simpler version of the problem that you can do, and see how solving it helps you analyse the original problem.

In this case, deciding how to convert numbers up to ten digits long into words is too difficult for an on-sight solution, so we split the number into groups of three digits – called 'bites' here – counting leftwards from the end of the number, as you do when you put commas into a number. Each bite is simply a number between 000 and 999 followed by the appropriate scaling word, such as 'thousand' or 'million' or (North American) 'billion'.

From a partial solution (a routine which can handle numbers from 0 to 999) it is a short step to a more general solution, because larger numbers can be rendered into words as

'0–999' BILLION
'0–999' MILLION
'0–999' THOUSAND
(AND) '0–999'

where a billion equals 1E9. So once we can translate the '0–999' we can

handle numbers up to 10 digits by applying the routine for 3-digit numbers up to four times. (There are other points to attend to, but that is the gist of it.)

Expressing a three-digit number in words then depends upon isolating the Hundreds, Tens and Units in turn, considering special cases – such as zero – and using the digits themselves as pointers into string arrays of words. These name the digits 1 to 9, the teens 10 to 19, and the tens 20 to 90. The scaling words stored are hundred, thousand, million and billion. In addition there are the words 'minus' and 'zero'.

All these words are stored in string arrays, which are filled in the routine from line 2000 forwards. You may wonder how they got there. If you have never seen READ and DATA statements before (as on lines 1000 to 1100), don't worry: they are the first topic of the next chapter.

5.4 Exercises

1. Write a program to read in a string of any length, and to return it padded or truncated to a given length. For example, if the inputs were

"Nice.guys.finish.last" 20

then the output would be

"Nice.guys.finish.las"

and if the inputs were

"Nice.one.Cyril" 20

then the output would be

"Nice.one.Cyril. . . . . "

where the dots represent blank spaces.

2. Rewrite the above procedure so that a third input indicates whether padding spaces should be leading or trailing (i.e. inserted at the front or the end of the output string).

3. Write a subprogram to input any string and return it converted to the case of its first letter. For example:

"Include me out" becomes "INCLUDE ME OUT",
"dyslexia Rules -- KO?" becomes "dyslexia rules -- ko ?".

4. Write subprograms to simulate the action of **LEFT$()** and **RIGHT$()**, using only **MID$**. It may be helpful to know that **MID$()** can be called with only two arguments (not three, as is normal). When it gets only two arguments, as in **MID$(a$,4)**, it returns the whole remaining string **(a$)** from the position specified by the second argument (here, 4).

5. Write a program to accept a string of any length, and to print out a frequency analysis of the characters it contains. How many times does '**a**' appear, how many times does '**b**' appear, how many blanks are there, and so on?

In a long passage of English text, the most common letters, in descending order of frequency, are ETAONIRSH; but the space character occurs even more often than 'e'.

6. Write a subprogram to simulate the action of **INSTR()**. When you have made that work, enhance it so that the question mark (**?**) acts as a 'wild card'. That is to say, it matches any character; so that "**B?S?**" would match "**BUSY**", "**BASE**", "**BOSS**", "**BUST**" and so on.

7. Rewrite Exercise 5 to report on the frequency of all two-letter combinations in the input string. Thus the string "Rewrite Exercise 5" contains re, ew, wr, ri, it, te, x . . . etc. For this purpose, ignore the distinction between upper and lower case. Also ignore spaces and punctuation signs.

6

Input/output

COMMUNICATING WITH THE MACHINE

The whole purpose of computing is input and output. Nobody would go to all the expense of buying and running a computer system if in return for your input you didn't get some worthwhile output. Everything that intervenes between the two processes is simply a means to an end. When you start reading computer books and magazines, however, you can miss that truth entirely: the impression is created that what really counts is how slick your programming is, how quickly your program works, how little space your code takes up. This is, of course, the voice of the computer buff speaking, for whom the point of computing is computing, period. Questions of worth, point and relevance are not considered because they are not asked. We can all sympathize with this blinkered view – *peccavi, peccabo* – but most people expect something more from their screens or printers than mere intellectual stimulation.

Ideally, all program design should start with the question of input and output. What do you want the output to be and how should it be presented? And, similarly: in what form is the input to be expected, and how should it be interpreted? As we shall see, Amstrad Basic permits a wide range of output forms and input methods: one of the arts of programming is choosing the techniques appropriate to the particular problem, so that input is obtained from the user in a natural and painless way, and the layout of the output is pleasant and conducive to understanding.

6.1 Keyboard input

The elementary forms of the INPUT instruction have been explained in previous chapters; and the example below illustrates some of their strengths and weaknesses.

Listing 6.1 Input demonstration

```
10 REM ******************************
11 REM ** Listing 6.1 :            **
12 REM ** INPUT DEMONSTRATION      **
15 REM ******************************
60 :
100 REM -- Plain Vanilla INPUT statements:
110 INPUT aged
120 INPUT "NAME ",name$
130 INPUT "ADDRESS ",address$
140 INPUT "NATIONALITY ",country$
150 INPUT "PLACE OF BIRTH ",plob$
160 PRINT
170 PRINT "Hello : ";name$;" aged ";aged
180 PRINT "You're ";country$;" born in ";plob$
190 PRINT "Now living at "
200 PRINT address$
220 END

run
? 38
NAME Brian MORRIS
ADDRESS 23, Padua Road
?Redo from start
ADDRESS 23 Padua Road
NATIONALITY British
PLACE OF BIRTH Beckenham, Kent
?Redo from start
PLACE OF BIRTH Kent

Hello : Brian MORRIS aged  38
You're British born in Kent
Now living at
23 Padua Road
Ready
```

INPUT is very useful for getting words and numbers from the keyboard. It is best accompanied by a prompt string, unless the nature of the expected input is made clear to the user in some other way. The trouble with INPUT is that it causes a complete halt in processing; and it is also vulnerable to faulty input. The former weakness is not relevant here, but the latter is crucial: the place of birth as given was "Beckenham, Kent" but this was rejected by the computer with a "?Redo from start" message. The comma between "Beckenham" and "Kent" acted as a delimiter on the input, whereas it was meant as part of the data. Furthermore, in line 110 the expected data is a number, but there is nothing to stop the user from entering some non-numeric string, in which case the variable **aged** would be set to zero as a result. We can deal with both these problems by making the following changes.

Listing 6.2 Input validation

```
10 REM ******************************
11 REM ** Listing 6.2 :            **
12 REM ** INPUT VALIDATION         **
15 REM ******************************
60 :
100 REM -- Simple Data-checking:
101 aged=0
105 WHILE aged<=0 OR aged>120
110   INPUT aged$
112   aged = VAL(aged$)
115   WEND
120 INPUT "NAME ",name$
130 LINE INPUT "ADDRESS ",address$
140 INPUT "NATIONALITY ",country$
150 LINE INPUT "PLACE OF BIRTH ",plob$
160 PRINT
170 PRINT "Hello : ";name$;" aged ";aged
180 PRINT "You're ";country$;" born in ";plob$
190 PRINT "Now living at:"
200 PRINT address$
220 END

Ready
run
? 37
NAME Richard FORSYTH
ADDRESS 23, Genoa Road
NATIONALITY Brit.
PLACE OF BIRTH London, England.

Hello : Richard FORSYTH aged  37
You're Brit. born in London, England.
Now living at:
23, Genoa Road
Ready
```

Here we see the LINE INPUT instruction for the first time, on lines 130 and 150 so that input to the variables **address$** and **plob$** may contain leading spaces and commas as part of the data.

The WHILE/WEND loop in lines 105 to 115 is a validation loop (of a sort), and is executed until a positive number greater than zero is given as the user's age. This illustrates several important points.

(1) You can check the data for errors even when you do not know exactly what the data should be: you can check the range of data, as here; you can check its data type, as in a way we do here, since the test on line 105 will reject any input that does not at least begin with numeric characters; you can check its 'picture', which we do not do here: the picture is a useful COBOL idea meaning the appearance of the data in terms of its maximum and minimum number of characters, position of decimal point, number of words, leading/trailing zeros, position and kind of delimiters, and so forth.

(2) This sort of validation cannot guarantee accurate input. The user can still lie or make a mistake. But it can help to ensure sensible data. In this program, for example, the user cannot get away with entering a negative or non-numeric age.

(3) It is usually easier to input data as a string, validate it as described, and then convert it into its intended form: strings are much easier to manipulate than numbers for such purposes, mainly due to the predefined string functions (described in the last chapter).

(4) The best validation in the world is no substitute for an informed and motivated user. Our example gives no hint of the kind of data expected in the **aged$** loop, nor any sign of whether the data has been accepted or rejected, nor any kind of error message.

Of these four points, the most important is the fourth. The other three are important tactical/technical points, but number four is strategic and it should be at the heart of any program design. If your input and output methods make life easy and pleasant for the user, then he or she will make life easier for you: errors will be less frequent and less complicated to trap and handle; program usage will improve and expand. It should not be necessary really to make this point, not when computers are nearly 40 years old and microcomputers have celebrated their tenth birthday, not when ergonomics and industrial design are venerable scientific disciplines, but bad design seems destined to survive us all. We should live so long!

6.2 READ, DATA and RESTORE

The READ and DATA statements are so convenient that we sneaked them into the example program of Chapter 5. Now it is time to explain how they work.

READ is used to assign to a list of variables the values obtained from one or more DATA statements. Neither statement can be used alone without the other. READ causes the variables listed to be given, in order, the next available data items from the collection of DATA statements. Program lines which begin with the word DATA are storage areas within the program, and as such are ignored by the Basic interpreter – rather like REM lines. In effect, they form an unnamed array. When a READ is encountered, this data area supplies the next available number or string. If the program attempts to read beyond the end of the data, it is halted with a "DATA exhausted" message.

It is important to realize that all the DATA statements in a program, taken together, constitute one long stream of data. When a READ needs a new value, the next item is taken from this stream. So it is the order, not the exact position, of items in DATA statements that matters.

The location of DATA statements is arbitrary so long as they occur in the correct order: a common practice is to collect them all together and place them just before the END. (If execution reaches a DATA statement, nothing happens: control passes on to the next program line.)

A READ statement takes the form

READ list

where 'list' is a sequence of variables or array elements separated by commas. A DATA statement takes the form

DATA list

where the list is a sequence of constants (not variables or general expressions) separated by commas. It is clearly neater to write

```
100  DIM m$(12),days(12)
120  FOR m%= 1 TO 12
140     READ m$(m%),days(m%)
160     NEXT
200  DATA January,31,February,28,March,31
220  DATA April,30,May,31,June,30
240  DATA July,31,August,31,September,30
260  DATA October,31,November,30,December,31
```

than to spell out twenty-four assignments such as

```
100  LET m$(1)="January" : days(1) = 31
110  LET m$(2)="February" : days(2) = 28
```

. . . and so on. (It is also quicker.)

Note that the type of READ variable, numeric or string, determines what kind of DATA item Basic expects to find. String data does not have to be in quotes, unless it contains commas or trailing spaces; and numeric constants can be read into string variables (as characters). But numeric variables should not be made to accept string data: try changing the last 31 on line 260 to "THIRTY-ONE" if you want to see what happens when you create a type mismatch.

A third instruction, RESTORE, gives the READ/DATA pair greater versatility. Its form is

RESTORE [linenumber]

where the 'linenumber' is optional. The effect is to set Basic's internal pointer, which keeps track of which DATA item to READ next, back to the start of the data stream (if no line number is specified) or to a particular line (if one is given). Thus

RESTORE

re-sets the data pointer to the very beginning, while

RESTORE 5500

re-sets it to start scanning from line 5500 of the program onwards. This is useful if you want to break up the DATA into chunks and read different chunks in different circumstances. Some Basics allow the linenumber to be specified by an arithmetic expression, but in Locomotive Basic it must be a constant.

Listing 6.3, below relies on READ and DATA statements to perform a kind of screening process. Its framework is quite general: it matches data objects against specified criteria, and keeps those objects that pass the tests. But for our example we have particularized it to deal with party invitations.

Listing 6.3 Party invitations

```
10 REM ******************************
11 REM ** Listing 6.3 :             **
12 REM ** PARTY INVITATIONS         **
15 REM ******************************
60 :
100 REM -- General-purpose Data Matching Program:
110 PRINT "Guest-processing Program:"
120 DIM proplist$(20)
130 zz$="****" : REM stop-code
140 PRINT
150 REM -- Main Line:
160 GOSUB 3000 : REM read in criteria
170 GOSUB 4000 : REM perform matching
200 PRINT "Bye for now!"
220 END
999 :
1000 REM -- Socialite Database:
1001 REM -- First the people:
1010 DATA Jessica
1020 DATA rich,jogs,under35,owns pony
1030 DATA Frances
1040 DATA under35,handsome
1050 DATA Jane
1060 DATA celebrity,jogs
1070 DATA Kate
1080 DATA rich,megastar,owns mercedes,owns land
1090 DATA Jo
1100 DATA handsome,celebrity,under35,hair dark
1110 DATA Anna
1120 DATA hair dark,megastar
1130 DATA Suzy
1140 DATA hair fair,under35,dumb
1150 DATA Mary
1160 DATA snobbish,poor,handsome,under35
1170 DATA Eleanor
1180 DATA hair dark,owns aircraft,jogs,under35
1190 DATA Sara
1200 DATA rich,handsome,under35,megastar
1210 DATA Emma
1220 DATA poor,dumb
1230 DATA Joan
```

```
1240 DATA celebrity,rich
1250 DATA Pauline
1260 DATA handsome,under35,owns winebar,jogs
1270 DATA Annalise
1280 DATA rich,jogs,ugly,under35
1290 DATA Lynette
1300 DATA hair fair,dumb,owns mercedes,under35
1310 DATA Jasmine
1320 DATA hair dark,handsome
1330 DATA Cynthia
1340 DATA snobbish,rich,hair dark,celebrity
1500 DATA Eddy
1510 DATA under35,male,handsome
1520 DATA Ivan
1530 DATA male,handsome,hair dark,megatsar
1540 DATA Carl
1550 DATA dumb,rich
1560 DATA Richard
1570 DATA male,dumb,poor,jogs
1580 DATA Joel
1590 DATA male,hair fair,under35
1600 DATA Fredrick
1610 DATA megastar,male,ugly,snobbish
1620 DATA William
1630 DATA male,under35,owns land,rich
1640 DATA Eric
1650 DATA male,handsome,rich,ugly
1660 DATA Andy
1670 DATA male,owns mercedes,rich,owns land
1680 DATA Paul
1690 DATA male,megastar,handsome,poor
1700 DATA Alan
1710 DATA male,hair dark,under35,celebrity
1720 DATA Maurice
1730 DATA male,dumb,rich,jogs,owns land
1740 DATA Sven
1750 DATA male,evil,megastar,snobbish
1760 DATA Mike
1770 DATA male,poor,handsome,celebrity
1780 DATA Charles
1790 DATA male,rich,snobbish,celebrity,owns land,hair dark,ugly
1800 DATA "****"
2000 REM -- Now the criteria:
2010 DATA FAILURE, poor,male
2020 DATA FAILURE, poor,NOT handsome
2030 DATA FAILURE, dumb,NOT hair fair
2040 DATA FAILURE, male,jogs
2050 DATA SUCCESS, handsome,NOT male
2060 DATA SUCCESS, rich,celebrity
2070 DATA SUCCESS, megastar
2080 DATA SUCCESS, under35,owns land,NOT ugly,NOT dumb
2090 DATA "****"
2100 REM -- end-of-data signal.
2110 :
3000 REM -- Criterion input routine:
3010 DIM criteria$(40)
3020 RESTORE 2000
3030 READ a$ : c%=0
3040 WHILE a$ <> zz$
3050   c%=c%+1
3060   criteria$(c%)=a$
3070   READ a$
3080   WEND
```

```
3090 PRINT c%;"items in criterion list."
3100 criteria%=c%
3110 RETURN
3120 :
4000 REM -- Matching routine:
4010 RESTORE
4020 READ name$
4030 IF name$=zz$ THEN RETURN
4035 selected%=0
4040 WHILE ASC(name$)<=ASC("Z") AND ASC(name$)>=ASC("A")
4050   GOSUB 4400 : REM read attributes
4060   suitable%=0
4070   GOSUB 5000 : REM match attributes
4080   IF suitable%>0 THEN GOSUB 6000 : REM do something with it
4090   name$=item$ : REM next name
4100   WEND
4110 PRINT selected%;"successful candidates."
4120 RETURN
4130 :
4400 REM -- Routine to get attributes:
4410 READ item$: features%=0
4420 WHILE item$<>zz$ AND ASC(item$)>=ASC("a")
4430   features%=features%+1
4440   proplist$(features%)=item$
4444   READ item$
4450   WEND
4470 RETURN
4480 :
5000 REM -- Single-match routine:
5010 c%=0 : suitable%=0
5020 WHILE c%<criteria% AND suitable%=0
5030   c%=c%+1
5040   c$=criteria$(c%)
5050   crit$="a"
5060   mismatch%=0
5070   REM -- now loop through tests:
5080   WHILE ASC(crit$)>ASC("Z")
5090     c%=c%+1
5100     crit$=criteria$(c%)
5110     negative%=INSTR(crit$,"NOT")
5120     IF negative% THEN crit$=MID$(crit$,5)
5130     GOSUB 5500 : REM test all attributes
5140     IF c%>=criteria% THEN crit$="AA"
5150     WEND
5160   IF c$="SUCCESS" AND mismatch%=0 THEN suitable%=1
5170   IF c$="FAILURE" AND mismatch%=0 THEN suitable%=-1
5180   IF c%<criteria% THEN c%=c%-1
5190   WEND
5200 RETURN
5220 :
5500 REM -- Routine to test one criterion against all properties:
5510 IF crit$="SUCCESS" OR crit$="FAILURE" THEN RETURN
5520 hits%=0
5530 FOR a%=1 TO features%
5540   IF crit$=proplist$(a%) THEN hits%=hits%+1
5550   NEXT a%
5560 IF hits%=0 AND negative%=0 THEN mismatch%=mismatch%+1
5570 IF hits%>0 AND negative%>0 THEN mismatch%=mismatch%+1
5580 RETURN
5590 REM -- mismatches if "NOT xx" found or "xx" not found.
5600 :
6000 REM -- Routine to process the selected candidate:
6010 PRINT #8: selected%=selected%+1
```

```
6020 PRINT #8, "Dear ";name$
6030 PRINT #8
6040 PRINT #8, "  I would like to invite you to my party"
6050 PRINT #8, "on the 31st of October in Zilog Hills,"
6060 PRINT #8, "because you are so utterly:"
6070 FOR a%=1 TO features%
6080   PRINT #8, proplist$(a%)
6090   NEXT
6100 PRINT#8: PRINT #8, "See you there!"
6110 PRINT #8: RETURN
6120 REM -- This routine is the one to be customized for other
            applications.
6130 :
```

```
Dear Frances

  I would like to invite you to my party
on the 31st of October in Zilog Hills,
because you are so utterly:
under35
handsome

See you there!


Dear Kate

  I would like to invite you to my party
on the 31st of October in Zilog Hills,
because you are so utterly:
rich
megastar
owns mercedes
owns land

See you there!


Dear Jo

  I would like to invite you to my party
on the 31st of October in Zilog Hills,
because you are so utterly:
handsome
celebrity
under35
hair dark

See you there!


Dear Anna

  I would like to invite you to my party
on the 31st of October in Zilog Hills,
because you are so utterly:
hair dark
megastar

See you there!
```

```
Dear Mary

  I would like to invite you to my party
on the 31st of October in Zilog Hills,
because you are so utterly:
snobbish
poor
handsome
under35

See you there!


Dear Sara

  I would like to invite you to my party
on the 31st of October in Zilog Hills,
because you are so utterly:
rich
handsome
under35
megastar

See you there!


Dear Joan

  I would like to invite you to my party
on the 31st of October in Zilog Hills,
because you are so utterly:
celebrity
rich

See you there!


Dear Pauline

  I would like to invite you to my party
on the 31st of October in Zilog Hills,
because you are so utterly:
handsome
under35
owns winebar
jogs

See you there!


Dear Jasmine

  I would like to invite you to my party
on the 31st of October in Zilog Hills,
because you are so utterly:
hair dark
handsome

See you there!


Dear Cynthia

  I would like to invite you to my party
```

```
on the 31st of October in Zilog Hills,
because you are so utterly:
snobbish
rich
hair dark
celebrity

See you there!


Dear Fredrick

  I would like to invite you to my party
on the 31st of October in Zilog Hills,
because you are so utterly:
megastar
male
ugly
snobbish

See you there!


Dear William

  I would like to invite you to my party
on the 31st of October in Zilog Hills,
because you are so utterly:
male
under35
owns land
rich

See you there!


Dear Sven

  I would like to invite you to my party
on the 31st of October in Zilog Hills,
because you are so utterly:
male
evil
megastar
snobbish

See you there!


Dear Charles

  I would like to invite you to my party
on the 31st of October in Zilog Hills,
because you are so utterly:
male
rich
snobbish
celebrity
owns land
hair dark
ugly

See you there!
```

The setting is that Jake and Candice Yupwards plan to throw a jet-set party to announce their arrival in Beverly Hills. Naturally they only want to invite 'suitable' guests. The problem is to decide who is suitable. After much heated debate, they agree that this vetting process should be settled objectively – with the help of their personal computer.

Candice enters the data on their friends and acquaintances (lines 1000 to 1800) in terms of such vital characteristics as whether they jog regularly ("jogs"), whether they own a winebar ("owns winebar") and suchlike. Meanwhile Jake formalizes their criteria of social acceptability (in line 2000 forwards). These criteria are stated in terms of what sort of people to reject out of hand (preceded by "FAILURE"), and what sort of people to invite (preceded by "SUCCESS").

Two examples should give you the flavour of the acceptance/rejection specifications. The line

```
2020 DATA FAILURE, poor,NOT handsome
```

means that any person with attribute "poor" who also lacks the attribute "handsome" will be rejected without further ado. On the other hand the line

```
2080 DATA SUCCESS, under35,owns land,NOT ugly,NOT dumb
```

can be interpreted as stating that anyone who is under 35, owns land, is not ugly and is not dumb should be invited.

To be technical, the items within a SUCCESS or FAILURE test are 'conjunctive' (they are ANDed, so all must be true); while the test themselves are 'disjunctive' (they are ORed and the first one to succeed determines the result). The NOT operator is provided to invert any particular attribute, as in "NOT male" for instance. Thus the essential Boolean operators of NOT, AND and OR are all provided, though in an unconventional manner. Because the criteria are tested sequentially, re-ordering can make a difference. For example, Jake has decided that all megastars deserve an invitation, but also that male joggers should be excluded (since they will bore the assembled company by talking about groin strains and knee injuries). However, since the male-jogger test on line 2040 comes before the megastar test on line 2070, a male megastar who jogs will be excluded. A more sophisticated version of this program would detect such conflicts and inform the user about them.

(Ivan fails to receive an invitation, by the way, because he is a Mega-Tsar, not a megastar.)

This is a rather frivolous example, but the program itself is a useful general-purpose pattern-matcher. Since conditional pattern-matching is at the heart of 99% of all large Artificial Intelligence systems we could claim that we are now doing Artificial Intelligence programming in good old Locomotive Basic. (But we won't.)

Once you have grasped the format of the 'facts' (data about persons) and the 'rules' (data about criterion tests) you can have some fun adapting it to other domains, such as purchasing decisions. But first try adding and altering a few of the tests, on lines 2000 onwards.

The subroutine that needs to be customized for different applications is the one starting on line 6000. Here it prints a stylized invitation to the **Name$** selected, but it could not do anything with it. Note incidentally that this routine, as written, introduces a new form of the PRINT statement, i.e. PRINT#8 . . . (e.g. on line 6030). The hash-sign followed by a channel or 'stream' number is used to direct output to a particular device. In Amstrad Basic channel 8 always refers to the printer, so this is the way of getting printed output, as distinct from output displayed on the screen. We consider channel numbers further in the next two sections.

6.3 Formatted output

The ordinary PRINT statement is satisfactory when you just want the results and are not too worried about their appearance: it removes from the programmer the burden of deciding exactly how the output should look. Eventually, however, a situation will arise where you want to control precisely which characters should appear in what position on each output line – just how many digits should be printed after a decimal point, and so on.

You can go some way towards formatted output with the aid of TAB and SPC, which are pseudo-functions that can be inserted into a PRINT list. TAB tabs across to a particular position on the output line, while SPC generates a given number of spaces. For example,

```
PRINT TAB(7);"Value of a is";SPC(7);a
```

prints the phrase "Value of a is" at character-position 7 on the line and the value of the expression **a** itself seven spaces later – starting at tab-position 26, in fact. But for complete control, you need the PRINT USING instruction.

The PRINT USING statement lets the format of an output line be governed by an image of that line in the program. Its format is

```
PRINT [#chan,] USING form; list
```

where 'chan' is the channel to which the output is directed (0 or absent for the screen, 8 for the printer), 'form' is a format specification, and the 'list' is a list of expressions to be printed. The list can contain numeric and character values, separated by commas and/or spaces, as in the normal PRINT instruction.

The format specifier is a string. Most of its characters will be printed as they stand, but some of them, including the hash-sign (#) and the ampersand

(&) have special significance. They are used to described 'fields' in the output. The following revision of a short program from Chapter 4 shows a simple format string composed of two numeric field specifiers.

Listing 6.4 PRINT USING with numbers

```
10 REM ******************************
11 REM ** Listing 6.4 :            **
12 REM ** PRINT USING WITH NUMBERS **
15 REM ******************************
50 :
55 DEF FNroundup(numb,dplaces%)=INT(numb*10^dplaces%+0.5)/10^dpl
aces%
60 :
75 form$="  ##          ##.##########"
100 PRINT " No. of Dec.  Rounded Value"
110 PRINT " Places       of e"
115 e = EXP(1) : REM base of natural logarithms
120 FOR n%=0 TO 10
130   PRINT USING form$; n%,FNroundup(e,n%)
140   NEXT n%
150 PRINT
160 END
```

```
 No. of Dec.  Rounded Value
 Places       of e
   0          3.0000000000
   1          2.7000000000
   2          2.7200000000
   3          2.7180000000
   4          2.7183000000
   5          2.7182800000
   6          2.7182820000
   7          2.7182818000
   8          2.7182818300
   9          2.7182818300
  10          2.7182818300
```

Notice that the same value appears whether printed with 8, 9 or 10 decimal places. This is because Basic cannot represent floating-point numbers with more than nine significant figures.

6.3.1 Numeric field specifications

Numeric fields in the image act as templates for the format of numeric expressions in the output list. They employ the hash sign (#), the decimal point (.) and the up-arrow in groups of four (∧ ∧ ∧ ∧). Each hash sign stands for a digit; the decimal point indicates the position of the decimal point, if any; and the four up-arrows, if present, reserve space for an exponent field – which always takes four character positions. Thus the value of 123.456 would be printed in the various ways shown below with the following different field specifications.

Template	Output
# # #	123
###.###	123.456
##.##^^^^	12.35E+01

Notice that automatic rounding (up or down as required) takes place and that numbers are justified to the right. Trailing zeros are added if necessary so that there are as many digits after the decimal point as hash signs in the image field.

An extra hash sign will be required to reserve space for the minus sign if the quantity output is negative. If insufficient space is reserved for a number, a percent sign (%) is printed as a warning in the first position and the field widened to accommodate it (thus throwing the formatting out of alignment).

If a numeric field is prefixed by two or more asterisks (**) then the number is printed with leading asterisks filling any unused positions. This is intended for cases where a number must be protected so that additional digits cannot be added at the front later (e.g. for cheques). A solitary asterisk is treated as a printing character. Numbers output in this mode should not have an exponent.

If a field begins with two or more pound signs (**££**) or two or more dollar signs (**$$**) the number is preceded on output by a 'floating' currency symbol. No spaces are left between the currency sign and the number. Thus the field **$$**##.## would cause 123.456 to appear as **$123.46** and 1.23456 to be printed as **$1.23**. An isolated dollar or pound sign is treated as a printing character. Numbers with floating currency symbols may not contain an exponent specification.

A single comma (,) placed just before the decimal point specifies that digits to the left of the decimal point are to be grouped in threes, for thousands. Thus (putting these various facilities together) the template ****######,.## with the number 123456.789 would cause ***$123,456.79** to appear.

6.3.2 String field specifications

Strings may also appear in the output list of a PRINT USING statement. If they do they should correspond, reading left to right, with a string field specification. These are composed with three special characters.

The exclamation mark (!) specifies that only the first character of the string is to be printed.

A pair of back-slashes enclosing zero or more spaces (\ \) specifies that

only the first *n* characters of the string are to appear, where *n* is the number of spaces inside the back-slashes plus 2. Strings that are shorter than *n* characters are left-justified within the field – i.e. padded with trailing blanks.

An ampersand (**&**) indicates that the whole string is to be printed 'as is', neither truncated nor padded to fit a fixed field width.

No format template for a string may exceed 255 characters in length.

Thus the following lines

```
2000 form$ = "THIS \ \ IS ! &MAT \ \FICATION"
2020 PRINT USING form$; "FIELDS","ARE","FOR","SPECIES THAT GRAZE"
```

would produce

```
THIS FIELD IS A FORMAT SPECIFICATION
```

as output.

6.4 Scanning the keyboard

Earlier in this chapter INPUT was criticized for causing a pause in program execution and for being vulnerable to faulty data – the latter being a consequence of the former. Your program cannot do anything about what is being typed in response to the INPUT prompt until RETURN is pressed, by which time it may be too late. The alternatives are the **INKEY** and **INKEY$** functions, both of which return the result of single keypresses and, therefore, do not require RETURN as the input delimiter.

The function **INKEY(n)**, where **n** is an integer expression, interrogates the keyboard to report which keys are being pressed. The keyboard is scanned fifty times a second. The example below

```
10 PRINT "Hit the Space Bar please."
20 IF INKEY(47) = -1 THEN GOTO 20
30 PRINT "You pressed the Space Bar, at last."
40 CLEAR INPUT
```

detects when the space bar is depressed, then prints a message and ends the program. The CLEAR INPUT instruction is used to flush out any remaining characters that the operating system is holding in the 'input buffer', a kind of queue for input that allows you to type ahead of the program, if you can go that fast.

INKEY(n) responds in the following way to the status of key number **n**.

INKEY value	SHIFT	CONTROL	Specified key
−1	–	–	UP
0	UP	UP	DOWN
32	DOWN	UP	DOWN
128	UP	DOWN	DOWN
160	DOWN	DOWN	DOWN

Thus you can test for joint presses of a key and the SHIFT or CONTROL keys.

The numbers used to identify keys in the **INKEY()** function are hard to remember (they do not correspond to the ASCII code), but fortunately they are embossed on the side of the machine, above the disc drive – at least on the CPC 6128.

The **INKEY$** function is the string version of the keyboard scanner. It works rather differently from **INKEY()**. It returns a one-character string, reflecting the key that is currently being pressed. If no key is pressed, the null string "" is returned. We actually used it in Listing 5.2.

INKEY$ requires no argument, and is often used in an implied loop, as below

```
40 k$=INKEY$ : IF k$="" THEN GOTO 40
```

so that it waits till a key is pressed. In certain applications, however, there is no need to wait. In a video game, for instance, the main program can scan the keys to see what the user is doing, and if he or she is doing nothing, it can get on with moving spacecraft or hungry monsters around the screen. This provides a degree of 'real-time' capability on your humble home computer.

6.5 MENUS and COMMANDS

Conversational computing (which is what Basic is good at, and the reason people put up with Basic's many faults) is all about communication between man and machine. Up to now all our programs have communicated with the user in a question-and-answer fashion. This is a rather *ad hoc* style of interacting with the user, but it is possible to put the interaction within a more coherent framework.

It is usual to characterize interactive programs as either menu-driven or command-driven, which, in the extreme case, means that whenever the user has a choice of options either you tell him/her what they are, or you don't! Either you present a nice numbered list of options on the screen (the 'menu') and wait for the user to choose, or you let the program continue whatever it is doing until the user hits some sort of command key. It is rather like the difference between INPUT and INKEY().

Of course programs are rarely one type or the other: a common approach is to display a menu at the start of a process, then execute the process until a command key is pressed. If there are lots of command keys then one of them may be set aside for calling up the menu again: it is a good idea to provide such a 'Help' command in complicated programs, and always use the same keystrokes for accessing it – CONTROL/H for example, or f0, something easily remembered.

Often the choice between menus and commands is dictated by the nature of the process. If there are things like alien battle-cruisers zooming around on the screen, or lots of information to be displayed, a menu is probably a nuisance; if the display is static or easily reconstructed then a menu is not much trouble. Another criterion – surely the main one – is usability: simple processes or expert users imply a command-driven approach, while complex options or untrained users demand plenty of help.

There are almost as many variations on these twin themes as there are programs. A notably subtle example is the popular word-processing system, WordStar, which has a blend of the two approaches. Most of the commands in WordStar require at least two keypresses in sequence; if you can remember them then you can treat the program as entirely command-driven. On the other hand, you can choose to have a Help menu occupying part of the screen at all times, and, in addition to that, if you make an initial command keypress and then pause, the Help menu changes to show you what your second keystroke options are. Menus lead to menus, and extra help can usually be requested. Some deep thought went into the program's 'user image'. And, as you might expect, a lot of experienced users call it 'fiddly', while many beginners find it bewildering. There are no right answers, in computing as in life, and fair is just what blond-haired people are.

Let us swallow some of our own medicine and begin the development of a menu-based data entry program, where the input is essentially a form-filling exercise.

Listing 6.5 Screen-based form-filling

```
10 REM ******************************
11 REM ** Listing 6.5 :             **
12 REM ** SCREEN-BASED FORM-FILLING **
15 REM ******************************
50 :
60 MODE 2: BORDER 23
100 REM -- Generalized Menu-program:
110 GOSUB 1000 : REM initialization
120 FOR r%=1 TO recs%
130   GOSUB 2000 : REM get record
140   NEXT
150 PRINT: PRINT "Bye!"
160 END
```

```
170 :
500 REM -- Data section:
510 DATA 4,10
520 DATA FORENAME, C,2,2,16
530 DATA SURNAME, C,2,4,17
540 DATA YEARS-OLD, N,33,4,4
550 DATA COMPANY, C,2,6,25
560 DATA ADDRESS, C,2,7,25
570 DATA TOWN, C,2,8,28
580 DATA COUNTRY, C,2,9,25
590 DATA POSTCODE, C,2,10,10
600 DATA NATIONALITY, C,2,12,16
610 DATA PLACE-OF-BIRTH, C,33,12,16
660 :
1000 REM -- Initialization routine:
1010 READ recs%, nfields%
1020 DIM db$(recs%,nfields%)
1030 padding$=STRING$(80," ")
1040 dots$=STRING$(80,".")
1050 RETURN
1060 :
2000 REM -- Menu-based input routine:
2010 RESTORE 520
2020 CLS: GOSUB 2500 : REM draw box
2025 RESTORE 520 : REM re-read box labels
2030 PRINT TAB(12);"Record : ";r%;
2040 FOR f%=1 TO nfields%
2050   GOSUB 3000 : REM move to line
2060   GOSUB 4000 : REM get & test data
2065   IF ok%=0 THEN GOTO 2060 : REM re-try
2070   db$(r%,f%) = item$
2080   NEXT f%
2100 RETURN
2120 :
2500 REM -- Screen-setting routine:
2530 FOR sf%=1 TO nfields%
2540   READ tagname$, type$,h%,v%,size%
2550   d$=LEFT$(dots$,size%)
2560   LOCATE h%,v%
2570   PRINT tagname$;SPC(1);d$
2580   NEXT sf%
2590 LOCATE 1,1 : REM back to top left.
2600 RETURN
2610 REM -- now data needs to be RESTOREd.
2620 :
3000 REM -- Single-tag display routine:
3020 READ tagname$, type$, h%,v%,size%
3030 d$=LEFT$(dots$,size%)
3040 LOCATE h%,v%
3050 PRINT tagname$;SPC(1);d$
3060 LOCATE h%+1+LEN(tagname$),v%
3080 RETURN
3090 :
4000 REM -- Single data-entry routine:
4010 item$=""
4020 s%=0 : ok%=1
4030 WHILE s%<size% AND ok%=1
4035   s%=s%+1
4040   ch$=INKEY$: IF ch$="" THEN GOTO 4040
4050   ok%=1
4052   IF ch$=CHR$(127) THEN ok%=0: GOTO 4100 : REM erase line
4055   IF ch$=CHR$(13) THEN s%=size%: GOTO 4100
4060   IF type$="N" AND (ch$>"9" OR ch$<"0") THEN ok%=0
```

```
4070    IF ok%=0 AND ch$="-" AND s%=1 THEN ok%=1
4080    IF ok%=0 THEN PRINT CHR$(7); ELSE item$=item$+ch$
4090    PRINT ch$;
4100    WEND
4105 CLEAR INPUT
4110 item$=LEFT$(item$+padding$,size%)
4112 IF VAL(item$)=0 AND type$="N" AND LEFT$(item$,1)<>"0" THEN
ok%=0
4115 IF ok%=0 THEN LOCATE h%+1+LEN(tagname$),v%: PRINT d$;: LOCA
TE h%+1+LEN(tagname$),v%
4120 RETURN
4130 :
```

This program takes our input demonstration of Listing 6.2 considerably further. In fact it goes a long way towards the input section of a fully-fledged database program. Its aim is to let the user enter data about individuals (FORENAME, SURNAME and so on) by filling in a form on the screen.

Line 60 simply sets up an 80-character screen mode and paints the surrounding border a pale cyan colour. We will deal more fully with MODE and BORDER in Chapter 8.

The shape of the form itself is defined by DATA statements in lines 500 to 660. This gives considerable flexibility: if you do not like the layout, or want an extra field or two, then simply alter the DATA statements. Line 510 states that there are 4 records, each consisting of 10 items. Lines 520 onwards define the items. For instance, line 550 says that the COMPANY field is for character data (C), and should be placed two positions in from the left margin and six lines down from the top of the screen. The response given by the user should not have more than 25 characters. Thus each piece of information has its own, labelled, slot on the screen. They can be placed independently, as long as they do not over-write each other.

The main routine is on line 2000 forwards. It calls subroutine 3000, which displays the field name ready for input, and subroutine 4000 which gets the date using **INKEY$** on line 4040 and performs several validation checks. For instance, numeric fields must consist of digit characters, possibly preceded by a minus sign.

The kind of output generated by running this program is illustrated below.

Output from Listing 6.5

```
      Record : 1
FORENAME Richard S.....

SURNAME Forsyth .....        YEARS-OLD 999.
COMPANY Human Race .....
ADDRESS 23, Genoa Road .....
TOWN Edgeville.....
COUNTRY Airstrip Two .....
POSTCODE ZIP9999...

NATIONALITY European .....   PLACE-OF-BIRTH EARTH.....
```

The dots following each field-name give a visual indication of the length of that field. The DELete key can be used, under program control, to correct mistakes by going back to the start of the field.

A few more sophisticated facilities, such as being able to scroll around the screen at will (using the cursor keys) would turn this into a respectable database front-end. Some readers may care to take up the challenge.

6.6 Example program [XSYS]

The example program for this chapter takes some of the ideas in Listing 6.3 (pattern matcher) a step further and allows us to progress towards a simple Expert System. A good deal of mystique surrounds the subject of expert systems; but – looked at from sufficiently far away – any expert system is likely to be a program for 'structured selection'. That is to say, there will be a list of 'hypotheses' or 'theories' concerning the state of the world or the right action to take and a set of 'evidence' bearing upon the hypotheses. The task of the program is to match the evidence against the hypotheses and identify the best or most likely one, or at least to help its user to do so.

In a medical context (and medicine has proved a happy hunting ground for expert systems builders) the hypotheses might be disease diagnoses and the evidence would be patient symptoms. In a geological context (an even more profitable hunting ground) hypotheses might include whether the rock was gas-bearing or oil-bearing and the evidence would come from the various measures logged by a borehole probe.

The problem of structured selection, at the heart of most expert systems, boils down to a decision about which hypotheses are best and worst supported by the available evidence.

Medical example

Evidence: Fever, Spots, Headache, Sore-throat, Belly-ache, Dizziness, Loss-of-appetite, Vomiting . . .

Hypotheses: Tuberculosis, Chicken-pox, Arthritis, Influenza, Duodenal-ulcer, Gastric-cancer, Black-death . . .

Geological example

Evidence: Speed-of-sound, Gamma-ray-emission, X-ray-porosity, Radioactivity, Hardness, Colour, Depth . . .

Hypotheses: Shale, Sandstone, Granite, Limestone . . .

An expert is supposed to know which items of evidence are relevant to which hypotheses.

The point is that different pieces of evidence help to confirm, or refute, different hypotheses. Expertise lies in sifting the strengths and weaknesses of these (possibly conflicting) indicators. This is true in many domains.

Our example avoids medicine (like the plague), for the very good reason that it is an offence to set up in the UK as an unqualified medical practitioner, and we do not want disgruntled users who diagnose themselves wrongly suing us on that count! Let us take a safer example – how to choose a house. Our program is an estate-agent's assistant, which may help the estate agent to make slightly better judgements in the process of matching clients to properties. Thus the items of evidence are the customer's preferences, and the hypotheses are the houses and flats for sale.

The program listing is followed by a specimen of the kind of printout produced by running it.

Listing 6.6 Very basic expert system

```
10 REM ********************************
11 REM ** Listing 6.6 :              **
12 REM ** VERY BASIC EXPERT SYSTEM   **
15 REM ********************************
20 REM Copyright (c) 1986, Warm Boot Ltd.
50 :
60 MODE 1: BORDER 20
100 REM -- Unreal Estates:
110 GOSUB 1000 : REM initialization
120 GOSUB 1500 : REM welcome screen.
130 GOSUB 2000 : REM structured selection.
140 GOSUB 5000 : REM optional print-out
150 PRINT CHR$(7): PRINT "Bye for now!"
160 END
170 :
500 REM -- Data Dictionary:
501 DATA 8
502 REM no. of variables.
510 DATA COST,N,2
520 DATA BEDROOMS,N,1
530 DATA POSTCODE,C,1
540 DATA TYPE, C,1
550 DATA MODE, C,1
560 DATA C.H., C,1
570 DATA GARAGES, N,1
580 DATA MAINROAD, C,1
590 :
600 REM -- Now the data itself:
610 DATA St. Matthews Court
620 DATA cost=44500,bedrooms=2,type=flat,mode=lease,postcode=N10
,*
630 DATA Lorraine Mansions
640 DATA cost=54000,bedrooms=2,type=flat,postcode=N7,C.H.=yes,ga
rages=0,*
650 DATA Broomfield Avenue
660 DATA cost=63000,bedrooms=3,type=terraced,mode=freehold,C.H.=
yes,postcode=N13,*
670 DATA Harlaxton Drive
680 DATA cost=49950,bedrooms=5,mainroad=no,garages=1,type=semi,m
ode=freehold,postcode=NG2,C.H.=yes,*
690 DATA Mount Pleasant Cottages
700 DATA cost=51750,bedrooms=2,c.h.=yes,garages=0,postcode=N14,*
710 DATA St. Mary's Road
```

```
720 DATA cost=52500,c.h.=yes,bedrooms=3,garages=1,postcode=N9,mo
de=freehold,*
730 DATA Malvern Road
740 DATA cost=52950,bedrooms=3,c.h.=yes,bedrooms=3,type=terraced
,postcode=N8,*
900 DATA "**"
999 :
1000 REM -- Initialization routine:
1010 READ evidence%
1020 DIM vars$(evidence%),kind$(evidence%)
1024 DIM criteria$(evidence%),wt%(evidence%)
1025 DIM proplist$(24),op$(4)
1030 FOR e%=1 TO evidence%
1040   READ vars$(e%),kind$(e%),wt%(e%)
1050   NEXT e%
1055 crit%=0
1060 op$(1)="EQ": op$(2)="NE"
1070 op$(3)="LT": op$(4)="GT"
1080 cr$=CHR$(13) : sp$=CHR$(32)
1085 eq$="=" : half = 0.5
1088 star$="*": halt$="**"
1100 REM -- also define windows:
1110 WINDOW #1,1,20,1,11
1120 WINDOW #2,22,40,1,11
1130 WINDOW #3,1,20, 12,22
1140 WINDOW #4,22,40,12,22
1160 LOCATE 12,23
1170 PRINT "XSYS AT YOUR SERVICE"
1172 PRINT TAB(12); "No. of variables =";evidence%;
1175 FOR w%=1 TO 4
1177   PRINT #w%,"WINDOW";w%
1180   NEXT
1188 INK 2,2
1190 PAPER #1,3: PAPER #4,3
1195 PAPER #2,2: PAPER #3,2
1200 REM -- also count the data:
1210 READ a$: counter%=0
1220 WHILE a$<>halt$
1230   IF a$=star$ THEN counter%=counter%+1
1235   READ a$
1240   WEND
1248 theories%=counter%
1250 DIM goodness%(theories%),dist(theories%)
1255 PRINT TAB(12);"No. of hypotheses=";theories%
1260 RETURN
1280 :
1500 REM -- Instruction routine:
1505 RETURN
1515 REM no instructions yet!
1525 :
1600 REM -- Single key input routine:
1610 ch$=INKEY$: IF ch$="" THEN GOTO 1610
1620 IF ch$<>cr$ AND ch$<>sp$ THEN GOTO 1610
1630 RETURN
1640 REM -- with ch$=cr$ or ch$=sp$
1660 :
2000 REM -- Main Input routine:
2010 done%=0
2020 used%=0
2025 FOR h%=0 TO theories% : goodness%(h%)=0
2030   NEXT h%
2050 WHILE used%<evidence% AND done%=0
2060   GOSUB 2200 : REM pick variable.
```

```
2065   GOSUB 2500 : REM get preference
2066   GOSUB 3000 : REM confirm input.
2070   IF confirm%=0 THEN GOTO 2060
2075   GOSUB 3200 : REM match choices.
2080   GOSUB 4000 : REM update display
2100   WEND
2120 RETURN
2150 :
2200 REM -- Get-choice routine:
2210 w%=1
2220 CLS#w%
2222 ev%=0
2230 PRINT #w%,"Variables left:"
2240 FOR v%=1 TO evidence%
2250   LOCATE #w%,2,4
2255   IF kind$(v%)="X" THEN GOTO 2300 : REM skip used vars.
2260   PRINT #w%, vars$(v%);SPC(16)
2270   GOSUB 1600
2280   IF ch$=cr$ THEN ev%=v%: v%=evidence%
2300   NEXT v%
2320 IF ev%<>0 THEN RETURN
2330 REM -- othewise test for quitting.
2340 LOCATE #w%,1,7
2350 PRINT #w%, "SPACE to Go on:"
2360 PRINT #w%, "RETURN to Quit:"
2370 GOSUB 1600
2380 IF ch$=sp$ THEN PRINT #w%,"Try again.": GOTO 2220
2390 done%=1
2400 RETURN
2420 :
2500 REM -- Get operator and tval$
2505 IF ev%=0 THEN RETURN
2510 w%=2
2520 CLS#w%: PRINT #w%, "Pick Operator:"
2525 comp%=0 : c%=1
2530 cmax%=4
2540 IF kind$(ev%)="C" THEN cmax%=2
2550 WHILE comp%=0
2560   LOCATE #w%, 2,4
2570   PRINT #w%, op$(c%)
2580   GOSUB 1600 : REM get ch$
2600   IF ch$=cr$ THEN comp%=c%
2610   c%=c%+1: IF c%>cmax% THEN c%=1
2620   WEND
2630 REM -- cycle till choice made.
2640 REM -- Now get target value:
2650 tv%=0: t%=0: tval$=""
2655 WHILE tval$=""
2660   RESTORE 600: w%=3: CLS#w%: t%=0
2666   PRINT #w%, "Possible values:"
2670   WHILE tv%=0 AND t%<theories%
2680     t%=t%+1
2690     READ name$
2700     GOSUB 6000 : REM get proplist$
2710     FOR f%=1 TO features%
2720       LOCATE #w%, 2,4
2730       GOSUB 6400 : REM split "prop=value"
2740       IF lh$=vars$(ev%) THEN PRINT #w%,rh$;SPC(12) ELSE GOT
O 2780
2750       GOSUB 1600
2760       IF ch$=cr$ THEN tv%=f%: tval$=rh$
2780       NEXT f%
2800     WEND
```

```
2820   WEND
2850 RETURN
2880 :
3000 REM -- Confirmation routine:
3010 confirm%=1: IF done% THEN RETURN
3020 ev$=vars$(ev%)
3030 comp$=op$(comp%)
3040 w%=4: CLS#w%
3050 PRINT #w%, "Confirm Choice:"
3060 PRINT #w%
3070 PRINT #w%, ev$
3080 PRINT #w%, SPC(2);comp$
3090 PRINT #w%, tval$
3100 PRINT #w%
3110 PRINT #w%, "SPACE to go on:"
3120 PRINT #w%, "RETURN to quit."
3130 GOSUB 1600
3140 IF ch$=cr$ THEN confirm%=0
3150 RETURN
3160 :
3200 REM -- Matching routine:
3210 IF done% THEN RETURN
3220 RESTORE 600
3222 crit%=crit%+1
3224 criteria$(crit%)=ev$+sp$+comp$+sp$+tval$
3225 REM -- save criteria for printout.
3230 w%=4: CLS#w%
3250 dmax=-1E+33 : dmin=1E+33
3260 FOR h%=1 TO theories%
3270   dist(h%)=half: NEXT h%
3280 FOR h%=1 TO theories%
3290   READ name$
3300   GOSUB 6000 : REM get proplist$()
3310   FOR f%=1 TO features%
3320     GOSUB 6400 : REM split prop/val.
3330     IF ev$=lh$ THEN GOSUB 3600 ELSE GOTO 3360
3340     dist(h%)=d
3350     IF d > dmax THEN dmax=d
3355     IF d < dmin THEN dmin=d
3360     NEXT f%
3380   NEXT h%
3390 kind$(ev%)="X" : REM mark as used
3400 used%=used%+1  : REM and count
3404 RESTORE 600
3410 FOR h%=1 TO theories%
3420   gain%=0 : READ name$
3422   name$=LEFT$(name$,16)
3425   GOSUB 6000
3430   IF dist(h%) >= dmax THEN gain%=-1
3440   IF dist(h%) <= dmin THEN gain%=1
3450   goodness%(h%)=goodness%(h%)+gain%*wt%(ev%)
3460   IF gain%>0 THEN PRINT #w%, "++ ";name$;
3470   IF gain%<0 THEN PRINT #w%, "-- ";name$;
3500   NEXT h%
3505 PRINT #w%
3520 RETURN
3530 :
3600 REM -- Distance routine:
3610 IF kind$(ev%)="N" THEN GOTO 3700
3620 REM -- Character data:
3630 d=1
3640 IF rh$=tval$ THEN d=0
3650 IF comp$="NE" THEN d = 1-d
```

```
3660 RETURN
3670 REM with d=1 or d=0
3680 :
3700 REM -- Numeric data:
3710 diff = VAL(rh$) - VAL(tval$)
3720 IF comp$="EQ" THEN d=ABS(diff)
3730 IF comp$="NE" THEN d=-ABS(diff)
3740 IF comp$="GT" THEN d=-diff
3750 IF comp$="LT" THEN d=diff
3760 RETURN
3770 REM with distance from target.
3780 :
4000 REM -- Display routine:
4004 IF done% THEN RETURN
4010 w%=4
4020 PRINT #w%, "SPACE or RETURN to go on:"
4030 GOSUB 1600
4040 CLS#w%
4050 dmax=0: dmin=0
4060 FOR h%=1 TO theories%
4070   IF goodness%(h%)<dmin THEN dmin=goodness%(h%)
4080   IF goodness%(h%)>dmax THEN dmax=goodness%(h%)
4100   NEXT h%
4101 RESTORE 600
4110 PRINT #w%, "BEST choices:"
4120 FOR h%=1 TO theories%
4130   READ name$
4140   GOSUB 6000 : REM keep in step
4150   IF goodness%(h%)>=dmax THEN PRINT #w%, LEFT$(name$,17)
4160   NEXT h%
4170 RESTORE 600: PRINT #w%
4180 PRINT #w%, "WORST choices:"
4190 FOR h%=1 TO theories%
4200   READ name$
4210   GOSUB 6000
4220   IF goodness%(h%)<=dmin THEN PRINT #w%, LEFT$(name$,17)
4230   NEXT h%
4240 PRINT #w%, CHR$(7);
4250 FOR w%=1 TO 3: CLS#w%: NEXT w%
4255 RETURN
4260 :
5000 REM -- Printout routine:
5020 w%=1
5030 CLS#w%
5040 PRINT #w%, "Do you want a print-out ? ";
5050 y$=""
5060 WHILE y$<>"y" AND y$<>"n"
5070   y$=INKEY$: IF y$="" THEN GOTO 5070
5080   y$=LOWER$(y$)
5090   WEND
5100 PRINT #w%, y$
5110 IF y$="n" THEN RETURN
5120 REM -- else do a listing:
5130 ch%=8: GOSUB 5500
5140 PRINT #ch%,"LIST OF CHOICES:"
5150 PRINT #ch%
5160 RESTORE 600
5170 FOR h%=1 TO theories%
5175   READ name$
5180   PRINT #ch%: PRINT #ch%
5190   GOSUB 6000 : REM get proplist$
5200   PRINT #ch%, TAB(4);name$
5202   IF goodness%(h%)>=dmax THEN PRINT #ch%, TAB(4);"**BEST**"
```

```
5205   IF goodness%(h%)<=dmin THEN PRINT #ch%, TAB(4);"**WORST**
"
5220   PRINT #ch%
5230   FOR f%=1 TO features%
5240     GOSUB 6400
5250     PRINT #ch%, lh$;TAB(17);eq$;TAB(22);rh$
5260     NEXT f%
5265   PRINT #ch%, "Score";TAB(17);eq$;TAB(22);goodness%(h%)
5270   PRINT #ch%
5280   NEXT h%
5290 PRINT #ch%
5300 RETURN
5330 :
5500 REM -- Routine to print criteria:
5510 PRINT #ch%,"YOUR PREFERENCES WERE:"
5520 PRINT #ch%
5530 FOR c%=1 TO crit%
5540   PRINT #ch%, criteria$(c%)
5550   NEXT c%
5560 PRINT #ch%
5565 PRINT #ch%
5570 RETURN
5580 :
6000 REM -- Routine to get attributes:
6010 features%=0
6020 READ a$
6030 WHILE INSTR(a$,eq$)
6040   features%=1+features%
6050   proplist$(features%)=UPPER$(a$)
6060   READ a$
6070   WEND
6080 RETURN
6090 :
6400 REM -- Routine to split prop=val:
6410 p%=INSTR(proplist$(f%),eq$)
6420 lh$=LEFT$(proplist$(f%),p%-1)
6430 rh$=MID$(proplist$(f%),p%+1)
6450 RETURN
6460 REM -- lh$ is property; rh$ is value.
6480 :


YOUR PREFERENCES WERE:

C.H. EQ YES
COST LT 52500
BEDROOMS GT 3
POSTCODE NE N7
TYPE NE FLAT


LIST OF CHOICES:


   St. Matthews Court

COST            =    44500
BEDROOMS        =    2
TYPE            =    FLAT
MODE            =    LEASE
POSTCODE        =    N10
Score           =     0
```

```
   Lorraine Mansions
   **WORST**

COST              =     54000
BEDROOMS          =     2
TYPE              =     FLAT
POSTCODE          =     N7
C.H.              =     YES
GARAGES           =     0
Score             =     -2


   Broomfield Avenue

COST              =     63000
BEDROOMS          =     3
TYPE              =     TERRACED
MODE              =     FREEHOLD
C.H.              =     YES
POSTCODE          =     N13
Score             =      1


   Harlaxton Drive
   **BEST**

COST              =     49950
BEDROOMS          =     5
MAINROAD          =     NO
GARAGES           =     1
TYPE              =     SEMI
MODE              =     FREEHOLD
POSTCODE          =     NG2
C.H.              =     YES
Score             =      4


   Mount Pleasant Cottages

COST              =     51750
BEDROOMS          =     2
C.H.              =     YES
GARAGES           =     0
POSTCODE          =     N14
Score             =      1


   St. Mary's Road

COST              =     52500
C.H.              =     YES
BEDROOMS          =     3
GARAGES           =     1
POSTCODE          =     N9
MODE              =     FREEHOLD
Score             =      2
```

```
   Malvern Road

COST              =     52950
BEDROOMS          =     3
C.H.              =     YES
BEDROOMS          =     3
TYPE              =     TERRACED
POSTCODE          =     N8
Score             =      3
```

In any expert system the distinction between the *knowledge base* and the *inference engine* is fundamental. In our case the 'knowledge' is defined on lines 500 to 999. It has a very simple format. First comes the number of variables (on line 501) or items of evidence. Then follow the names of these variables (lines 510 to 580), each of which has a type (C or N) that indicates whether it describes character or numeric data and an integer which weights its relative importance. For example, on line 510, we have weighted **COST** as 2, while all other variables have a weight of 1. This means that price is twice as important as any other variable in reaching a decision.

On lines 600 onwards the actual data can be seen, describing a number of properties for sale. The first item is always the name of the property (or the road where it is situated). This is followed by attribute–value pairings, in no particular order. For instance

bedrooms=2, C.H.=Yes

signify, in turn, a property with two bedrooms and central heating. Not all the attributes need to be specified for every house: the program can cope with missing features. (You should be able to make sense of the rest of the details on your own.)

Each entry is ended by an asterisk (*), and the list of entries is ended with a double asterisk (**): see line 900. Upper and lower case letters are interchangeable.

Note particularly that it is only these data which tie the program down to an estate-agency application. You could easily remove the data shown here and plug in another 'knowledge base' – concerned with purchasing cars, say – to change the domain of application. (This is crucial in full-scale expert systems.)

The inference engine, such as it is, occupies much of the rest of the program. See especially lines 2000–2120 and 3200–3780. We have extended the ideas introduced in Listing 6.3. In essence, each hypothesis is scored according to how well it matches each of the user's criteria. By examining the matching routine on lines 3200 to 3530 you will be able to see how the system takes care of missing attributes, how it scores hypotheses by their 'distance' from the desired specification, and so forth.

The moral of our story, however, is not concerned with expert systems as

such, but with input and output; and the message is that if you give the user an opportunity to make a mistake he will grab it with both hands and put his foot in it (so to speak). It is asking for trouble expecting a user to type lines like

MODE EQ LEASEHOLD
GARAGES GT 0

correctly, or even to remember that a field is called MODE or GARAGES (rather than, say, TENURE or GARAGE). The objective of the input subroutines, at lines 2200 and 2500, is therefore to restrict the user's choice to that which is valid – and nothing else.

How is this achieved? By taking advantage of the Amstrad's WINDOW facility and splitting the screen into four separate 'windows' or boxes. These are defined on lines 1110 to 1140. For example, the line 1120

1120 WINDOW #2, 22,40,1,11

puts window number 2 in columns 22 to 40 across and rows 1 to 11 down within the main screen – i.e. the top right-hand quadrant. A PRINT statement directed to this window, such as line 2570, writes the output in this portion of the screen, which acts – to all intents and purposes – like a miniature screen in its own right.

What happens when the user makes a preference is that each window-box is used to get one component of that preference. Suppose the preference is

COST LT 63000

indicating a target price of under £63000. This consists of three components – a variable name, an operator and a numeric value. In the top left box the user is presented with each of the (eight) variables one by one. Pressing the Space bar steps on to the next choice; pressing RETURN selects the variable shown. In the next box the user is stepped through the possible operators, EQ, NE, GT, LT in the same manner. Finally, in the lower-left box the user is presented with the actual values present in the data (for the variable chosen) rather than being required to type in a number. The fourth window is used for reporting intermediate results back to the user.

All the user ever does is type either SPACE or RETURN. Any other characters are simply ignored; yet this is sufficient to build up a list of quite sophisticated choices, such as that below.

COST	**LT**	**54000**
POSTCODE	**NE**	**N7**
C.H.	**EQ**	**YES**
MODE	**EQ**	**FREEHOLD**

The burden on the user's memory is minimal. He or she need not remember the precise names of the variables, nor guess what a sensible value for a

comparison is, nor even worry about whether the term LEASE or LEASEHOLD is used in the knowledge base.

Readers can have some fun modifying this program to their own requirements. The technically minded may be interested to know that it uses a modified 'forward chaining' strategy. A forward-chaining system works through a sequence of evidence (whereas a 'backward chaining' system works through a list of hypotheses). In this case the order of evidence is, by default, the order in which variables are named on lines 510 onwards, and can be changed by a straightforward program amendment. But the user does not have to deal with them in the order given. By stepping through the variables offered in Window 1 ("Pick Variable:"), you can pick any variable not yet used. So the evidence can be dealt with in any order you like.

6.7 Exercises

1. Write a reaction time program. It should warn the user that input is expected of him or her, and then flash something onto the screen which the user must copy correctly. The system variable TIME can be used, or the program can keep its own counter, but the user must be given some measure of the time taken to respond. The degree of accuracy should also be measured. This program can be regarded as the first step on the road to a typing tutor.

Is INPUT appropriate for this exercise, or should one of the single-keypress functions be used?

2. Write the missing welcome-screen routine for Listing 6.6, from line 1500 onwards. This should give the user instructions.

3. Calculate and print an interest table for investments showing the return at rates from 1% to 12% over from 1 to 25 interest periods, using the formula

R = (1+I/100) ∧ N

Where **R**=return, **I**=Interest rate in percent and **N**=number of periods. Employ PRINT USING for the output.

4. Write a monthly billing program for a time-sharing computer bureau where charges to users are £6 per hour of connected time; 10 pence per minute of runtime; 8 pence per disc block used; plus a standing charge of £10 per month. Data to be read are: account-name (a string); connect-time in hours; runtime in minutes; and average disc-block usage over the month. This information is to be held in the form of DATA statements.

5. Write a program that accepts a list of frequencies and produces a histogram from them. It will be easier if the histogram is printed sideways.

For instance, given the scores 7, 0, 4, 17, 2, 8, 10 the program would produce something like the output below.

```
01: *******
02:
03: ****
04: ******************
05: **
06: ********
07: **********
```

The final version of your program should scale the input frequencies so that they fit onto the printed page. If you give the WIDTH 80 command, line width is set to 80 characters, in which case the data should be scaled to take up no more than 75 stars.

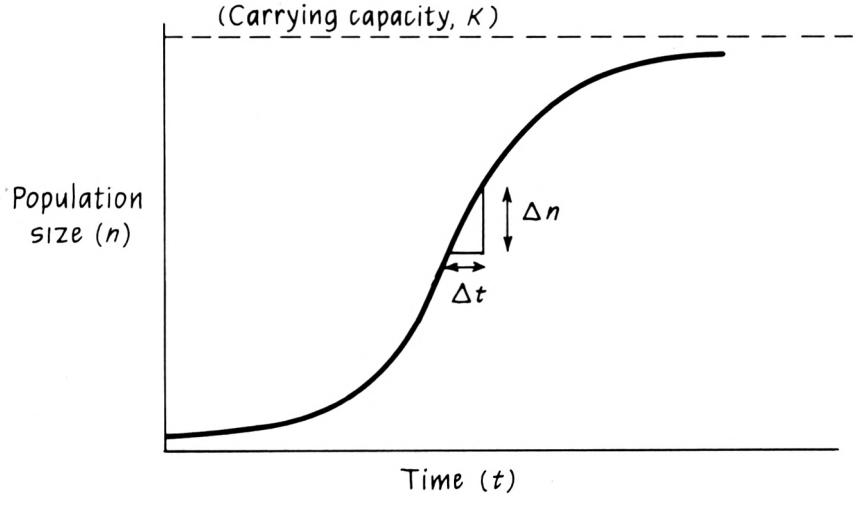
6. We have not yet started to delve into the world of the Amstrad's graphical facilities proper; but we have already found occasion to use the LOCATE instruction, whose form is

LOCATE [#chan,] horizontal, vertical

where 'horizontal' gives the displacement to the right and 'vertical' gives the displacement downwards – both in character-position units. (The optional 'chan' is a numeric expression in the range 0 to 7 referring to a screen window.) This statement can be useful for producing 'chunky' graphics.

Write a program using LOCATE to display the 'logistic curve' on the

Fig. 6.1 Logistic curve



screen. The logistic curve is used to describe a variety of growth processes, e.g. the growth of a population of bacteria in a supporting medium. It describes the rate of growth in terms of the formula

Dn = Dt * rate * n * (1 − n / K)

Where **Dn** is the increase in numbers during a short time-step and **Dt** is the time-step. The other variables are

rate the rate of increase (per time-step);
n the number of items (bacteria, say);
K the maximum carrying capacity of the medium.

Use a LOCATE to position a printed asterisk on the screen at the correct **X** and **Y** coordinates to plot the curve for 12 hours. You will find it necessary to calculate the values at least every quarter of an hour, even though they are only plotted once per hour – to maintain accuracy. Suitable test values are: **rate = 0.2**; **K = 1200**; with initial **n = 20**.

7

Files

ELEMENTARY DATA PROCESSING

The proprietor of a small firm arrives at his office one morning to find it gutted by fire. What is his first concern? The computer? No: that was insured. The software? No: that can be replaced. What he is really worried about is the security of his data files, because they are the lifeblood of the business.

If you wanted to sum up the difference between home computing and commercial data processing in one word, that word would have to be 'files'. Yet the Amstrad microcomputers – especially with disc drives – allow the home computer user a long way into the field of professional data processing, by virtue of their extensive file-handling facilities.

A file is a collection of information held for long-term storage on disc or tape. There are two main types of file – sequential and random-access. You have already met sequential files, since every time you SAVE a program you are creating a file in which that program is stored.

Sequential files can only be processed in serial order from the beginning. For example, a program is always LOADed starting at the first character and continuing with successive characters until the end.

Random-access files are divided into chunks known as 'records' each of which can be accessed individually. If you had 30 records in a random-access datafile you could read the 28th one, then the 15th, then write new data to the 17th, and so on in whatever order you pleased.

Magnetic tape, being an inherently serial medium, can only support sequential files. So those of you using CPC 464s without disc units will find yourselves left out in the cold for parts of this chapter. Please grin and bear it for the moment. Fortunately disc-drive prices continue to fall, so it may not be very long before the whole chapter is relevant to you. Then you too can enter a new realm of computing.

7.1 Simple file-handling

Files on the Amstrad microcomputers are handled by the Operating System (AMSDOS for Amstrad Disc Operating System). You have already met the

SAVE and LOAD filing commands and become familiar with the idea that each file has a name, and that a group of files is collected together in a catalogue.

With AMSDOS these ideas are somewhat extended. Each file is referenced by a 'file specification'. In its simplest form a file specification is simply a file name, e.g.

CAVERNS

consisting of up to eight characters. But the name may be followed by an 'extension' of up to three characters, as in

CAVERNS.BAS

which usually indicates the type of the file. So far we have always been using files as receptacles for Basic programs, so when we have given commands like

SAVE "BONK"

AMSDOS has very helpfully expanded the command to

SAVE "BONK.BAS"

to save us typing. You have probably noticed that when you give the CAT command, you get a list of filenames with their extensions. You may also have worked out that "BONK.BAK" (if such a file exists) is the back-up version of "BONK.BAS". When you SAVE a Basic program, AMSDOS stores the previous version, if any, with a .BAK extension – just as a precaution, in case you did not really mean to overwrite it.

A complete file specification also requires a drive prefix at the front (**A:** or **B:**) to select the disc unit. With a single-drive system there is only one disc unit (labelled **A**), so you can forget the drive prefix most of the time as it defaults to **A:** normally. But occasionally you may have a use for commands like

SAVE "B:ZONK.OLD"

which saves a file named ZONK with extension OLD on drive B.

Section 7.1.1 gives a list of the most useful commands for manipulating files. These are additional to the CAT, LOAD and SAVE commands already discussed.

All the AMSDOS commands that follow should be prefixed by the vertical bar character (|) which tells Basic to pass the command line over to the operating system (unlike CAT, LOAD and SAVE). This character can be found over the commercial at-sign key (@).

Sometimes you may want to refer to a group of files. For this purpose you can use the so-called 'wild cards', the question mark and the asterisk, in the

file specification. The question mark (?) matches any single character, while the asterisk (*) matches the remainder of the name or extension. Thus

|DIR,"B?N?.*"

could be used to match BONE, BONK, BUNK, BANG, BOND, BANK, BINS etc, with any extension at all.

7.1.1 AMSDOS commands

A

This sets the default drive to A, which is the main drive within the computer. On single-drive systems A is the only drive. This saves putting **A:** in front of all file references.

B

This sets the default drive to B. The main (or only) drive is A, which is selected at start-up time.

CPM

This switches to the alternative disc operating system, the widely-known CP/M (provided you have a CP/M disc in the selected drive). CP/M deserves a book of its own (and has got several). We cannot describe it in detail here. For more information consult Appendix C, or if that is insufficient buy one of the books.

DIR [,filespec]

This displays the directory, like CAT but in CP/M style. The 'filespec' is a string that specifies which files to show. If it is omitted, all files are listed. Thus **|DIR,"*.BAK" displays all files with BAK extension.**

DISC

Causes disc to be used for file input and output till the next TAPE command. This is the normal state if you have a disc unit.

ERA [,filespec]

Erases all files which match the specification. Thus **|ERA,"*.BAK"** deletes all .BAK files.

REN,newspec,oldspec

Renames a file. The old name is given second, and the new name first. Thus

|REN,"ZONK.ZAP","WHAM.WOW"

gives the file WHAM.WOW the new name ZONK.ZAP henceforward.

TAPE

Selects the cassete as the file input/output medium until the next DISC command.

USER,integer

Determines which of up to sixteen sections of the disc directory (numbered 0 to 15) is to be selected. Normally number 0 is selected, but this allows a disc to be partitioned between several applications for convenience.

(Note that an AMSDOS command is separated from its subsequent argument, if there is one, by a comma, not by a space.)

7.1.2 Basic file concepts

We have just presented a list of commands for doing various useful things with files from outside a program. In order to do any serious data processing we need a set of instructions for handling files from within a Basic program; and to make sense of those instructions it is important to understand a few essential file-handling concepts. The most important of these are: *record*, *field*, *file*, *OPEN*, *CLOSE*, *read* and *write*.

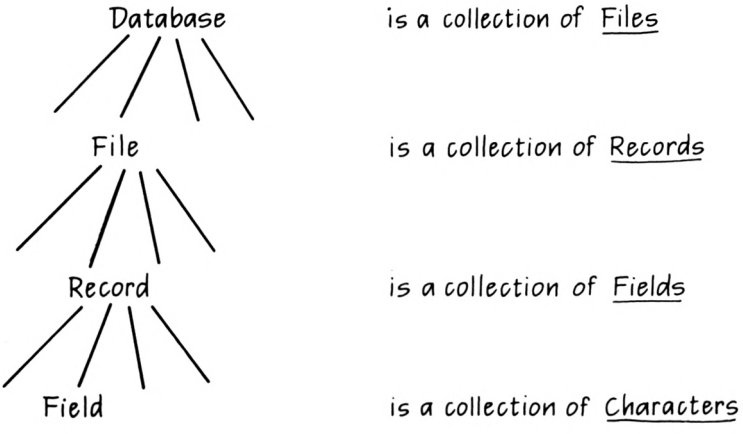
A *record* is the fundamental unit of file processing. It is a collection of information about an identifiable object. For instance, a record could contain the name, address and phone number of an individual; or it could contain the title, author, price, publisher and ISBN of a certain book; or again it could hold the temperature, rainfall and pressure readings of a given day. The essential point is that the data in one record belong together. The size of a record may be anything from a single character (extremely small) to thousands of characters (extremely large).

Records are normally considered to be composed of one or more *fields*. A field is an elementary data item such as a string or a number, e.g. the string giving a book's title in a bibliographic record, or the number of millimetres of rainfall in a weather record. Often one field is singled out as the 'Key Field' which uniquely identifies the record: this may be a name, a standard book number, an account number or some other identifying code.

A *file* is best viewed simply as a group of records. Incidentally, a 'database', if you have come across that term, is the next level up – being a grouping of logically related files. The hierarchical diagram in Fig. 7.1 illustrates this.

If a program is going to use a file it is necessary to inform the filing system by OPENing that file; and when the program finishes with it, it should be CLOSEd. Closing a file ensures that certain routine housekeeping tasks which keep things in order for future use are performed properly.

Fig. 7.1 Datafile hierarchy



Finally, the object of the exercise is to *read* and/or *write* data on the file. Data are normally read or written one record at a time. Reading transfers information from the file into program variables in memory, where computations can be performed in the usual way. Writing transfers results obtained within the program to a file for long-term storage.

7.1.3 Channels and files

Within a program a file is referred to by a channel number, not by its name. Before reading or writing a file, therefore, the program must set up correspondence between a file name and a channel number. This is done in one of two ways, by OPENIN or by OPENOUT.

Thus

OPENIN "MARCH85"

opens the file MARCH85 for reading. The file must already exist.

Likewise

OPENOUT "APRIL85"

opens APRIL85 for writing. This file will be created if it does not already exist. If a file of that name does exist, its contents will be erased, so this statement demands careful handling.

Amstrad Basic is extremely mean in its allocation of channel numbers for files. In fact, there is only one – number 9! This means that you can only have one input and one output file open at a time. (Looking on the bright side,

though, it also makes it hard to forget which channel number is for disc access!)

Channel numbers in Amstrad Basic are allocated as follows:

0 normal screen input and output;
1–7 screen 'windows' as defined by the programmer (see Chapter 6);
8 output to the printer (see Chapter 6);
9 disc file access.

A PRINT or INPUT statement with no channel specified defaults to channel zero – the display screen.

7.1.4 Sequential read and write

Once a file has been opened, data may be read from it by the INPUT# statement. Thus

```
1000 INPUT #9, maxtemp,mintemp,rainfall,sunshine
```

would read four (floating-point) numbers from a disc file into the variables **maxtemp**, **mintemp**, **rainfall** and **sunshine**. It is essential therefore that the data has been recorded in numeric form. The syntax of this statement consists of the keywords INPUT, then the channel selector (#9), followed by a list of one or more variables separated by commas.

You can also use LINE INPUT #9, to get a complete line of data, commas and all, in a similar fashion.

To write data sequentially to a file opened with OPENOUT, you can use the PRINT# statement. So

```
2200 PRINT# 9, weekday$,humidity,pressure,D%+1
```

would place a string and three numbers onto the file connected to channel 9. The final number would be an integer, with the value of **D% + 1**, which shows that the list of items to be written need not be variables, they can be expressions as well – of string, floating-point or integer type.

However, the PRINT#9 statement formats the output just as a normal PRINT would do on the screen. This is fine if the output is for human consumption; but if you want to write data which will later be read back into a program, something else is required. For this the WRITE statement should be used. Thus

```
2200 WRITE #9, weekday$,humidity,pressure,D%+1
```

would be more realistic than the previous version of line 2200. This is because WRITE puts strings in quotation marks and separates data items with commas, for reading back later.

Actually the concept of records, which we stated was so important in file-handling, is not fully supported by Amstrad Basic. The record structure has to be imposed by the programmer. That is to say: you must decide on the structure of your file and stick to it. If you put out the data as follows

```
2000  FOR R% = 1 TO 36
2010     WRITE #9, R%name$(R%),item(R%),code%
2020     NEXT R%
```

then you must read it back in a similar way. The data as written will, in this case, consist of groups of four items (or fields). The first field is a number, an integer in fact, **R%**; the second is string data, the **R%**th element of the array **name$()**; the third item is a floating-point number from the array **item()**, and the fourth one is another integer value. To read such data back with

```
2500  FOR Q% = 1 TO 36
2510     INPUT# 9, position%(Q%),name$(Q%),item(Q%),codenum%(Q%)
2520     NEXT Q%
```

would make sense, since it preserves the order integer, string, floating-point, integer; but the following would be erroneous

```
3500  FOR Q% = 1 TO 36
3510     INPUT# 9, name$ (Q%),item(Q%),R%
3520     NEXT Q%
```

for two reasons. The first problem is that the data were written in groups of four items, but has been read back three items at a time. The other problem is that the first item in the INPUT# list is a string, the second a floating-point number and the third an integer. This does not correspond to the order of data on file, and so a "Type Mismatch" error will occur.

The simplest way to avoid this kind of mistake is to establish a record structure in your program plan (e.g. string, string, integer, floating-point – or whatever is appropriate) and always ensure that you WRITE# and INPUT# in accordance with that structure.

It is also worth knowing that Amstrad Basic datafiles are held in ASCII format: they are merely sequences of characters. This means that you can usefully employ a simple dodge to create a sequential file without any programming at all. All you have to do is use AUTO as if you were typing in a Basic program and enter the data, line by line, using commas as separators. When you have finished use a special form of the SAVE command, followed by the letter A, which saves in ASCII mode. Thus

```
SAVE "TEST.DAT",A
```

saves the current 'program' (which is actually data) in ASCII format as the file TEST.DAT on disc. Having done so, you can list the program and edit it in the normal way, using the cursor-control and COPY keys for editing if desired.

You can also read it by a Basic program, as long as you remember to skip over the line numbers at the front of each line (which are held as characters too).

For example the following listing shows part of a datafile created in this way for a simple sales ledger.

Listing 7.1 Sales datafile

```
[Listing 7.1:]

50 16 JAN84,16 JAN84,BIS/COLLINS,20,HULK,23,3,SOFT,Q1,S
60 25 JAN84,25 JAN84,CENTURY/BROCKBANK,21,HULK,23,3,SOFT,Q1,S
70 27 JAN84,30 MAY84,R. OLNEY,22,AI84,110.4,14.4,SEMI,Q1,S
80 8 MAY84,30 MAY84,R. OLNEY,22,AI84,-64.4,-8.4,SEMI,Q2,S
90 27 JAN84,10 FEB84,SINCLAIR/SEARLE,23,AI84,220.8,28.8,SEMI,Q1,S
100 28 JAN84,30 JAN84,BRAINSTORM,24,HULK ROYALTIES,37.95,4.95,SOFT,Q1,S
110 28 JAN84,9 APR84,STC/ELLAM,25,AI84/ES84,211.6,27.6,SEMI,Q1,S
120 28 JAN84,1 MAR84,DHSS,26,ES84,202.4,26.4,SEMI,Q1,S
130 28 JAN84,20 MAR84,PROTEK,27,ES84,202.4,26.4,SEMI,Q1,S
140 30 JAN84,8 MAR84,CAD CENTRE,28,AI84,124.2,16.2,SEMI,Q1,S
150 30 JAN84,21 MAY84,BEESGREEN/LAWRENCE,29,AI84,124.2,16.2,SEMI,Q1,S
160 31 JAN84,5 MAR84,BRIGHTON POLY,30,AI84,124.2,16.2,SEMI,Q1,S
170 2 FEB84,2 MAR84,ATARI/NORLEDGE,31,AI84,220.8,28.8,SEMI,Q1,S
180 2 FEB84,5 MAR84,NIBKARN/SLATTERY,32,AI84,110.4,14.4,SEMI,Q1,S
190 2 FEB84,17 FEB84,FAIRCHILD,33,AI84,303.6,39.6,SEMI,Q1,S
200 2 FEB84,17 FEB84,FAIRCHILD,34,ES84,285.2,37.2,SEMI,Q1,S
210 16 FEB84,22 FEB84,FAIRCHILD,34,ES84,-41.4,-5.4,SEMI,Q1,S
220 2 FEB84,27 FEB84,BOLTON INSTITUTE,35,AI84,138,18,SEMI,Q1,S
230 4 FEB84,28 FEB84,PA TECHNOLOGY,36,ES84,101.2,13.2,SEMI,Q1,S
240 8 MAY84,8 MAY84,PA TECHNOLOGY,36,ES84,-82.8,-10.8,SEMI,Q2,S
250 4 FEB84,11 FEB84,QDQ SYSTEMS,37,ES84,101.2,13.2,SEMI,Q1,S


[Sales Datafile (extract).]
```

Here there are ten items (or fields) per line: invoice date, date of payment, name of customer, invoice number, brief description, gross value, VAT value, type-code, VAT quarter and S (for sales, because P is used for purchases on another file). Incidentally, lines 210 and 240 show negative sales: these are credit notes or repayments. Accountants are so hooked on what Goethe called the 'sublimest creation of the human mind' – namely, double-entry book-keeping – that they refuse to acknowledge the invention of negative numbers. Still, why force your computer to behave like a mediaeval clerk?

Datafiles in the format shown in Listing 7.1 can be read into a Basic program by a couple of lines such as the following.

```
1010 INPUT #9, invdate$,paydate$,name$,code$,item$
1020 INPUT #9, sumtotal,vatvalue,heading$,quarter$,type$
```

But remember that the program should strip off the line-number from the front of **invdate$** before processing it. This is quite easy to arrange.

The only snag with this short-cut method is that if you use words which are Basic keywords, such as 'to', 'time', 'for' etc., the interpreter will put them into upper case – whether you want it or not – because it thinks it is dealing with a program.

7.1.5 End of file

After opening and reading or writing a file it is important to close it. Among other things closing a file writes a special marker – the end-of-file mark – in the correct place so that the end of the file can be detected later.

The statement

```
CLOSEIN
```

closes the input file on channel 9, if open. The statement

```
CLOSEOUT
```

will close any open output files.

It is not always possible to know in advance how big a file is, so the function of EOF is provided to detect the end-of-file marker and avoid attempts to read beyond the data. Thus

```
1250 IF EOF THEN GOSUB 6000 : REM shut up shop
```

tests whether the input file currently open has any more data left to be read. If it has not EOF will be true and the shut-down routine will be executed.

We can now put these ideas together to show you a small sequential file processing example.

7.1.6 Sequential file creation

This little example has a meteorological flavour. It assumes that we want to store (and later access) the London weather readings for a particular month. In this case the month is March 1984. Only the first 15 days are actually shown, to save space.

Once the data is on file, we could write a variety of programs to do all sorts of things with it – even things we had not thought of at the time of storing the data. For instance, we could plot the rainfall pattern, count how often sunny days were also windy, and so on. The list is endless.

Listing 7.2 Sequential file creation

```
10 REM ******************************
11 REM ** Listing 7.2 :            **
12 REM ** SEQUENTIAL FILE CREATION **
15 REM ******************************
60 :
100 REM -- Reads DATA into file:
110 INPUT "Which File "; f$
120 OPENOUT f$
130 RESTORE
140 READ counter%
150 FOR r% = 1 TO counter%
160   READ dayname$,mintemp,maxtemp,humidity
170   READ rainfall,maxgust,sunshine,mb%
180   WRITE #9, dayname$,mintemp,maxtemp,humidity,rainfall,maxgust,sunshine,mb%
190   NEXT r%
200 CLOSEOUT
210 PRINT "File ";f$;" created."
220 PRINT counter%;" records written."
222 PRINT "BEWARE the IDES of MARCH!";CHR$(7)
250 END
300 REM -- Weather Data for file:
310 DATA 15
320 DATA THU01, 4.5,13.2,59,1.6,23,4.4,1015
330 DATA FRI02, 5.1,7.3,42,1.1,62,5.6,1008
340 DATA SAT03, 3.2,7.3,51,0,50,4.6,1017
350 DATA SUN04, 2.7,8.5,72,0.1,20,0,1025
360 DATA MON05, 7.7,10.5,91,0.2,15,0,1030
370 DATA TUE06, 9.1,13.4,66,0,20,0.5,1036
380 DATA WED07, 8.0,10.8,65,0.02,28,0.1,1037
390 DATA THU08, 2.4,6.6,55,0,27,0.3,1040
400 DATA FRI09, 3.6,6.9,57,0,24,1.2,1039
410 DATA SAT10, 4.1,8.3,76,0.5,20,0,1030
420 DATA SUN11, 5.7,10.6,59,0.2,22,0.4,1027
430 DATA MON12, 4.2,8.6,74,3.9,25,1.5,1020
440 DATA TUE13, 2.5,6.7,60,0.02,31,0.4,1022
450 DATA WED14, 3.0,10.1,51,0,24,8.5,1015
460 DATA THU15, 1.8,6.6,70,0,29,0.7,1015

Ready
run
Which File ? march84
File march84 created.
 15  records written.
BEWARE the IDES of MARCH!
Ready
```

7.1.7 Sequential file listing

One of the simplest things we can now do with this file is to list its contents on the screen (or on the printer if requested). It is not a very glamorous task, but it is such a common requirement that it is useful to see it done. After all, the first thing you need to know about a datafile is whether it contains what you think it should.

Listing 7.3 Sequential file listing

```
10 REM ******************************
11 REM ** Listing 7.3 :            **
12 REM ** SEQUENTIAL FILE LISTING  **
15 REM ******************************
60 :
100 REM -- Reads data from file:
101 ouch%=0 : REM 8=printer, 0=screen.
```

```
110 INPUT "Which File "; f$
120 OPENIN f$
130 IF EOF THEN PRINT "Empty File!": END
135 MODE 2: WIDTH 80: ZONE 10
140 PRINT #ouch%
150 PRINT #ouch%,"Day","Min. C","Max. C","Humidity",
160 PRINT #ouch%,"Rain","Maxgust","Sunhours","Pressure"
180 WHILE NOT EOF
190   INPUT #9, date$,minc,maxc,damp,rain,wind,sunshine,pressure%
200   PRINT #ouch%, date$,minc,maxc,damp,rain,wind,sunshine,pressure%
220   WEND
250 CLOSEIN
300 END
```

Day	Min. C	Max. C	Humidity	Rain	Maxgust	Sunhours	Pressure
THU01	4.5	13.2	59	1.6	23	4.4	1015
FRI02	5.1	7.3	42	1.1	62	5.6	1008
SAT03	3.2	7.3	51	0	50	4.6	1017
SUN04	2.7	8.5	72	0.1	20	0	1025
MON05	7.7	10.5	91	0.2	15	0	1030
TUE06	9.1	13.4	66	0	20	0.5	1036
WED07	8	10.8	65	0.02	28	0.1	1037
THU08	2.4	6.6	55	0	27	0.3	1040
FRI09	3.6	6.9	57	0	24	1.2	1039
SAT10	4.1	8.3	76	0.5	20	0	1030
SUN11	5.7	10.6	59	0.2	22	0.4	1027
MON12	4.2	8.6	74	3.9	25	1.5	1020
TUE13	2.5	6.7	60	0.02	31	0.4	1022
WED14	3	10.1	51	0	24	8.5	1015
THU15	1.8	6.6	70	0	29	0.7	1015

7.2 Random-access to the RAM-bank

AMSDOS does not provide genuine random-access to files; but it does allow you (on the CPC6128) to simulate random-access files using a spare 64K of memory known as the RAM-Bank.

7.2.1 Getting into the bank

Before you can use the RAM-bank as a kind of file, you must run a program provided on the system disc called BANKMAN (the 'Bank Manager'). Then you can use the extra commands BANKOPEN, BANKREAD, BANKWRITE and BANKFIND.

To start using a file in random-access mode you first have to copy it from disc into memory. (And don't forget to copy it back afterwards to disc if you have made changes.) Then you can treat the memory-resident copy as a random-access file.

When regarded as a file, the 64K in the RAM-bank is divided into a number of records. Each record has a fixed length of 1 to 255 characters, though 2 is recommended as a minimum. To establish the record length, you put the command

|BANKOPEN,size

into your program. This, in effect, opens the RAM-file and sets the record

length to 'size' – an integer expression. You can now put data into the RAM-bank at any given location and get them out again from any location.

The RAM disc is primarily for random-access, but AMSDOS maintains a current-record pointer which is advanced by each read or write operation. This makes it easy to do sequential access too.

7.2.2 BANKREAD and BANKWRITE

To write data into the RAM-file, use the command

|BANKWRITE,@code,datatext [,position]

where the arguments are as follows:

code	is a variable to receive the AMSDOS error signal which is −1 if the record is beyond the end of the RAM-bank or −2 if something else goes wrong;
datatext	is the string data to be written;
position	specifies the record number.

The 'position' parameter is optional.

To read data from the RAM-file, use the command

|BANKREAD,@code,datatext [,position]

where the parameters are the same as for BANKWRITE. Thus

|BANKWRITE,@flag%,"Text",99

puts "Text" into the 99th record of the RAM-file and returns −1 or −2 in **flag%** if something goes wrong. Similarly

|BANKREAD,@flag%,text$,p%

gets the string data from position **p%** of the RAM-file (or sets **flag%** to −1 or −2 if it fails for some reason).

If the string data in a BANKWRITE command does not fill the whole record, old characters will be left lying around at the end of the record. If a BANKWRITE string is too long to fit in a record, it will be truncated to prevent it overflowing into the next record. The same applies, in reverse, with BANKREAD, which cannot extend the current length of its string argument (datatext). It is a good plan, therefore, to pad out or truncate your strings so that they are exactly the right length before using BANKREAD or BANKWRITE – i.e. the same size as the record-length specified in BANKOPEN.

Note that the record numbers are counted from zero, and that if you omit the position parameter, the next record will be selected. This makes sequential processing easier.

Note also that these are AMSDOS extended commands, not normal Basic instructions. You must

RUN "BANKMAN"

with side 1 of the system disc inserted prior to using them.

7.2.3 Finding your way around

Normally you will think of your files as composed of records; and, to keep things simple, the records will usually be of fixed length.

Thus if we refer back to our weather-data file in Section 7.1.7, each record contains the information relating to one day. The record structure is as follows.

Name	Type
dayname$	string
mintemp	floating-point
maxtemp	floating-point
humidity	floating-point
rainfall	floating-point
maxgust	floating-point
sunshine	floating-point
pressure%	integer

But AMSDOS insists that each record of the RAM-file is a single string. This makes it awkward to use the RAM-bank for collections of mainly numeric data.

One way round the problem is to use STR$ to turn numbers into strings and then pad them out to a fixed length. Then several numbers can be stored as several AMSDOS records. This means that more than one physical record will be used to hold one logical record. It is a little messy, but it works, as shown by the example program (Listing 7.4), e.g. line 1100.

7.2.4 Free banking

The BANKFIND command is useful for searching through the RAM-file, faster than would be possible in a Basic program. Its form is

|BANKFIND,@code,datatext [,firstrec [,lastrec]]

where 'code' is a variable to receive the AMSDOS return code and 'datatext' is the string being sought. The optional 'firstrec' and 'lastrec' parameters give the positions for the beginning and end of the search, if desired.

The return code will be the record number where the string was found, if

the search is successful. If the search fails it will return −3 if the string could not be found: −2 if there was a system error; and −1 if the starting record is beyond the end of file or greater than the ending record number.

Thus

```
|BANKFIND,@code%,"target",a%,a%+20
```

would scan RAM-file records **a%** to **a%+20** inclusive looking for the character string "target" and set the integer variable **code%** to the location where it was found, or to a negative number if it was not found.

7.3 Example program [GOLFERS]

Our example program implements a simple golf-club handicapping system, according to the standard scratch scoring scheme adopted in the UK in 1983. It is not a complex application, but it ought to give you an insight into practical data processing.

The main file contains details of golfers and their handicaps. The record format is outlined below.

Field	Type
name$	string
handicap	floating-point

At the start of the program the data on file are copied into the RAM-bank for random-access processing (subroutine 1000). At the end they are copied out again (subroutine 3000).

To work correctly each player should have (or be given) a unique name.

The program can do four things:

(1) Create a new player;

(2) Delete an existing player;

(3) Show a player's handicap;

(4) Revise a player's handicap
(given that player's latest score).

You will see that the main line of the program is simply a loop that lets the user choose one of these options and calls the appropriate procedure to carry it out. (This example will only work with disc-based systems.)

For non-golfers, a few words of explanation may be useful. (Golfing freaks can skip straight to the listing.)

Golfers all have 'handicaps' to enable players of different strengths to compete together. A handicap is a number of strokes. If I have a handicap of

20 and go round in 85 strokes and you have a handicap of 10 and go round in 80 then I beat you (on handicap) by 5 strokes.

The handicap is the number of strokes above 'par' which the player is expected to require. Thus if par for a course is 72 and your handicap is 12 then you would be expected to take 84 strokes to play a round on that course.

There are strict rules governing the revision of handicaps which attempt to ensure fair play. These are incorporated into the program below. Essentially your handicap goes down when you do well and up when you do badly. However it tends to go down faster than it goes up!

If you play worse than expected (i.e. your actual score is more than par plus your handicap) you add 0.1 to your handicap if it is less than 5.5; otherwise you add 0.2 to it. So a player with an exact handicap of 7.8 who shot an 88 on a 75-par course (13 over par) would go up to 8. In fact, any score over 83 would raise the handicap by the same amount. Since exact handicaps are always rounded up to the nearest whole number for playing purposes, this rise from 7.8 to 8.0 would not affect the next game.

If you play better than expected, the system is a bit more complicated. Assuming your handicap is greater than zero, it will be adjusted downwards in the following manner.

First you work out the differential between your expected score (par + handicap) and the actual score. Let us say you shot 80 on a 72-par course and your handicap was 15. Your differential is 7 (= 15 − 8).

Now you look in the table below

Handicap category	Adjustment rate
0 to 5.4	0.1
5.5 to 12.4	0.2
12.5 to 20.4	0.3
20.5 and above	0.4

and subtract the adjustment figure for each stroke of the differential. Thus your handicap would go down by 7 * 0.3 from 15 to 12.9.

There is one further complication. When a player's handicap is reduced so that it goes from a higher category to a lower category (e.g. from 12.6 to 12.3), it is reduced at the rate for the higher category only so far as to bring it into the lower category; any remaining reduction is made at the rate for the lower category. Thus for a player on 13 who shoots 8 over par, giving a differential of 5, the first two strokes' deduction are at the 0.3 rate (down to 12.4) and the rest are at the 0.2 rate (down to 11.8). This is taken care of by a loop in the routine on lines 2570 to 2640 of the program.

Well, don't blame us: we didn't invent this scheme. Just park yourself at the 19th hole and leave it to the computer!

Listing 7.4 Golf-club handicapper

```
10 REM ******************************
11 REM ** Listing 7.4 :            **
12 REM ** GOLF-CLUB HANDICAPPER    **
15 REM ******************************
20 REM -- Must RUN "BANKMAN" first.
50 ON ERROR GOTO 500 : REM error trap
60 :
100 REM -- GOLFERS:
101 ouch%=0 : REM 8=printer, 0=screen.
110 MODE 1 : BORDER 15
120 recsize%=24
125 gaps$=SPACE$(recsize%)
130 chan%=9
140 INPUT "File Name "; f$
150 OPENIN f$
160 GOSUB 1000 : REM load data from file
170 PRINT: PRINT "Data from file : ";f$
180 command%=99
190 REM -- Main Loop:
200 WHILE command% <> 0
210   GOSUB 1200 : REM menu
220   IF command%=1 THEN GOSUB 1500 : REM new player
230   IF command%=2 THEN GOSUB 1700 : REM old player
240   IF command%=3 THEN GOSUB 2000 : REM show player
250   IF command%=4 THEN GOSUB 2300 : REM handicap
260   GOSUB 2200 : REM pause
270   WEND
280 CLOSEIN
300 GOSUB 3000 : REM write data back out
320 PRINT: PRINT "Bye!"
330 END
333 :
500 REM -- error trap section:
510 IF ERR=32 AND ERL=150 THEN GOSUB 600 ELSE ON ERROR GOTO 0
530 RESUME 150 : REM resume
550 :
600 REM -- create new file (1st time only):
605 n$=gaps$
610 OPENOUT f$
615 WRITE#chan%, n$,-99
620 CLOSEOUT
630 RETURN
650 REM contains one dummy record.
660 :
1000 REM -- Data input routine:
1010 recs%=0: r%=0
1020 |BANKOPEN,recsize%
1040 WHILE NOT EOF
1050   INPUT#chan%, name$,h
1060   h$=LEFT$(STR$(h)+gaps$,recsize%)
1070   name$=LEFT$(name$+gaps$,recsize%)
1080   |BANKWRITE,@r%,name$
1090   IF r%=-1 THEN PRINT "UGH!"
1100   |BANKWRITE,@r%,h$
1110   IF r%=-1 OR r%=-2 THEN PRINT "EH!?": STOP
1120   recs%=recs%+2
1130   WEND
1140 CLOSEIN
1150 PRINT recs%/2;"items read from file ";f$
1155 PRINT "Press any key to go on :"
```

```
1156 c$=INKEY$: IF c$="" THEN GOTO 1156
1160 CLEAR INPUT: RETURN
1170 :
1200 REM -- Menu subroutine:
1210 CLS: PRINT
1220 PRINT TAB(16);"Options:"
1230 PRINT TAB(16);"========"
1240 PRINT TAB(16);"[";f$;"]"
1250 PRINT
1260 PRINT "1 .... Add a new player"
1270 PRINT "2 .... Delete an old player"
1280 PRINT "3 .... Show a player's details"
1290 PRINT "4 .... Revise a player's handicap"
1300 PRINT: PRINT "No. of option (0 to quit) : ";
1310 c$=INKEY$: IF c$="" THEN GOTO 1310
1313 command%=ASC(c$)-48
1320 PRINT c$
1330 RETURN
1340 :
1500 REM -- New-player routine:
1510 PRINT: PRINT "Adding new player:"
1520 at%=-1 : p%=0
1530 WHILE at%<0 AND p%<recs%
1540   |BANKREAD,@r%,name$,p%
1550   |BANKREAD,@r%,h$,p%+1
1560   IF r%<0 THEN PRINT "Gasp": STOP
1570   IF VAL(h$)<0 THEN at%=p%
1580   p%=p%+2
1590   WEND
1595 IF at%<0 THEN at%=recs%: recs%=recs%+2
1600 REM -- Insert at at%:
1610 h=-99 : name$=""
1620 WHILE (h<0 OR h>36) OR name$=""
1630   LINE INPUT "Player's name: ", name$
1640   LINE INPUT "Handicap is  : ", h$
1650   name$=LEFT$(name$+gaps$,recsize%)
1660   h=VAL(h$)
1670   WEND
1680 GOSUB 4000 : REM write a record
1695 PRINT "inserted at location ";at%
1696 RETURN
1699 :
1700 REM -- Record-deletion routine:
1710 PRINT: PRINT "Record deletion:"
1720 GOSUB 5000
1730 GOSUB 4400 : REM show it
1740 PRINT "OK to delete it (Y=Yes) ";
1750 c$=INKEY$: IF c$="" THEN GOTO 1750
1760 ok%=(c$="Y") OR (c$="y")
1765 PRINT c$
1770 IF NOT ok% THEN PRINT "Not erased.": RETURN
1780 REM -- mark as dead with -ve handicap:
1788 PRINT
1790 h=-99
1800 GOSUB 4000 : REM write record
1810 PRINT "Player ";name$;" erased."
1820 GOSUB 2200 : REM delay
1850 RETURN
1860 :
2000 REM -- Show-details routine:
2010 PRINT: PRINT "Show details for player:"
2020 GOSUB 5000 : REM -- get record number
2030 IF at%<0 THEN PRINT "Sorry!": RETURN
```

```
2040 IF at%>recs% THEN PRINT "Sorry!": RETURN
2050 GOSUB 4400 : REM display routine
2060 GOSUB 2200 : REM wait
2070 RETURN
2080 :
2200 REM -- Delay routine:
2202 REM uses w%, w$
2210 w%=0 : w$=""
2220 WHILE w%<222 AND w$=""
2230   w$=INKEY$
2240   w%=w%+1
2250   WEND
2260 RETURN
2270 :
2300 REM -- New-handicap routine:
2310 PRINT: PRINT "Revision of handicap data:"
2320 GOSUB 5000 : REM get record
2330 GOSUB 4400 : REM display it
2340 PRINT "OK to alter handicap ? ";
2350 c$=INKEY$: IF c$="" THEN GOTO 2350
2360 ok%=(c$="Y") OR (c$="y")
2370 IF NOT ok% THEN RETURN
2380 IF h<0 THEN RETURN : REM dud record
2390 PRINT
2400 INPUT "Latest score was : ", s
2410 INPUT "Par for the course:", p%
2420 s = INT(s-p%) : REM difference from par
2430 GOSUB 2500 : REM compute new handicap
2440 PRINT: PRINT "New handicap for ";name$
2450 PRINT h;" (playing off ";INT(h+0.5);")"
2460 GOSUB 4000 : REM write it.
2470 PRINT "Press any key to go on:"
2475 c$=INKEY$: IF c$="" THEN 2475
2480 RETURN
2490 :
2500 REM -- Handicap-revision routine:
2510 d = s - INT(h+0.5)
2520 IF d=0 THEN RETURN
2530 IF d>0 AND h<=5 THEN h=h+0.1: RETURN
2540 IF d>0 AND h>5 THEN h=h+0.2: RETURN
2550 REM -- gets here if better than usual:
2560 IF h<=0 THEN h=0: RETURN
2570 WHILE d < 0
2580   x = 0.1
2590   IF h > 20.4 THEN x=x+0.1
2600   IF h > 12.4 THEN x=x+0.1
2610   IF h > 5.4 THEN x=x+0.1
2620   h = h - x
2630   d = d + 1
2640   WEND
2650 RETURN
2660 REM with h as new handicap.
2670 :
3000 REM -- Write-file routine:
3010 p%=0
3020 OPENOUT f$
3030 WHILE p%<recs%
3040   |BANKREAD,@r%,name$,p%
3050   |BANKREAD,@r%,h$,p%+1
3060   IF r%<0 THEN STOP
3070   h=VAL(h$)
3080   WRITE#chan%, name$,h
3090   p%=p%+2
```

```
3100   WEND
3110 CLOSEOUT
3120 PRINT recs%/2;" items dumped to ";f$
3130 PRINT
3140 RETURN
3150 :
4000 REM -- Write-record routine:
4010 REM puts name$,h at at%
4020 h$=LEFT$(STR$(h)+gaps$,recsize%)
4030 |BANKWRITE,@r%,name$,at%
4040 |BANKWRITE,@r%,h$,at%+1
4050 IF r%<0 THEN STOP
4060 RETURN
4070 :
4200 REM -- Read-record routine:
4210 REM gets name$,h from at%
4215 name$=gaps$
4220 |BANKREAD,@r%,name$,at%
4230 |BANKREAD,@r%,h$,at%+1
4240 IF r%<0 THEN STOP
4250 h=VAL(h$)
4260 RETURN
4270 :
4400 REM -- record-display routine:
4410 REM shows record at at%:
4420 GOSUB 4200
4430 IF h<0 THEN PRINT "EMPTY RECORD!": RETURN
4440 PRINT
4450 PRINT "Player name : ";name$
4460 PRINT "Handicap is : ";h
4470 PRINT "Playing off : ";INT(h+0.5)
4480 PRINT
4490 RETURN
4500 :
5000 REM -- Routine to get record-number:
5010 PRINT: ok%=0
5020 WHILE ok%=0
5030   INPUT "Record number "; n%
5040   IF n%>=0 AND n%<recs% THEN ok%=1
5044   IF n% MOD 2 = 1 THEN ok%=0
5050   IF ok%=0 THEN PRINT "Please try again!"
5060   WEND
5070 at%=n%
5080 RETURN
5090 :
```

The program structure is quite straightforward. This is a simple, menu-driven, file access program. The top-level procedures are as follows.

600 Initializes a file the 1st time it is used.
1200 Displays the user's options.
1500 Adds a new player's details to file.
1700 Deletes a player from the file.
2000 Displays details about any player.
2300 Revises a player's handicap after a game.

The file has room for up to 200 records. Since golf handicaps can never be below zero, a negative number is used to mark a record as empty. This is also used by subroutine 1700 to indicate a deleted record.

The lower level utility routines are as follows.

4200 Reads a given record from file.
4000 Writes out player details file.
4400 Shows a given record's contents on screen.
5000 Asks user for a selected record number (and checks for valid input).
2200 Introduces a delay, before clearing screen.

By encapsulating the file-handling in routines, we make it relatively easy to make alterations in the data structure. For instance, we could change the size of the records or the number of fields just by modifying subroutines 4000 and 4200.

If you want to see the clever bit – the part that updates players' handicaps – look at the subroutine on line 2500 onwards. It is worth noting how small the calculation section is, compared to the rest of the program which only exists to make those calculations obtainable. That is the way of data processing: the computational parts, although essential, form a trivial proportion of the eventual code.

7.4 Exercises

1. Using the March 1984 data shown in Section 7.1, write a program that reads such a file and calculates the averages for all its numeric fields. These are: **mintemp**, **maxtemp**, **humidity**, **rainfall**, **maxgust**, **sunshine** and **pressure%**. (The last one is an integer.)

Temperatures are in degrees Celsius; humidity is a percentage; rainfall is measured in mm; maximum wind gust is in knots; sunshine is in hours and pressure is in millibars. If you are keen, you can obtain more data from the London Weather Centre in High Holborn, WC1V 7HX.

2. Write a program to read a text file and count the number of characters and the number of words.

A word can be defined, for our purposes, as any sequence of letters, digits and hyphens beginning with a letter. You will probably find it convenient to have a logical variable called **ISINWORD%** which is FALSE to begin with and changes from FALSE to TRUE on encountering a letter (either in upper or lower case). It will change from TRUE to FALSE when the program reads a character other than a letter, a digit or a hyphen ('-'). On every transition from FALSE to TRUE, it should increment the word counter. Counting characters is easy.

FALSE and TRUE can be represented as 0 and 1.

This program could be useful if your word processor does not have an

automatic word-count facility – especially if, like us poor freelance journalists, you are paid by the word! If your word processor does have a word counter, you can compare them both on the same file and ponder why they give different answers. (They give the same results? I beg your pardon.)

3. Enhance the previous program by making it count sentences as well. You can define sentences how you like; but a sequence of a letter, one of '.', '?' or '!', and then a non-letter might do to start with as an indicator of the end of a sentence.

A further improvement would be to calculate and print the average number of characters per word and the average number of words per sentence.

4. It is often useful to have a menu program which runs another program (using the CHAIN statement) chosen by the user from a list of options. The program clears the screen and displays a list of options, rather like that below.

A. LIST Lists a data file.
B. SORT Sorts a data file into order.
C. KILL Deletes a data file.
D. ZONK Does something clever.

All the user has to do is press the single letter (A, B, C, D) corresponding to his or her choice and that program will be CHAINed into action.

Write a general-purpose menu program to do this task. It should allow up to 26 choices which are read from DATA statements. For the four options shown above, for instance, the data would be as listed below.

```
9000 DATA 4
9010 DATA LIST, Lists a data file
9020 DATA SORT, Sorts a data file into order
9030 DATA KILL, Deletes a data file
9040 DATA ZONK, Does something clever
```

5. Write a program to sort the records of the golf-club file into ascending order of player name. You can assume that there are no more than 200 entries on the file and read them all into a pair of arrays for sorting in memory. Once sorted, they can be written out to file again in the correct order. They could also be listed on the printer in alphabetical order, which would considerably ease the life of any golf-club secretary brave enough to try using this system.

A suitable algorithm is the Shell sort, for which a pair of routines are given below.

Listing 7.5 Shell sort

```
10 REM ******************************
11 REM ** Listing 7.5 :            **
12 REM ** SHELL SORT               **
15 REM ******************************
60 :
100 REM -- Shell-sort demonstration:
101 ouch%=8 : REM 8=printer, 0=screen.
110 MODE 1 : BORDER 15
120 INPUT "How many items "; n%
130 DIM name$(n%), h(n%)
140 FOR i%=1 TO n%
150   a$=CHR$(65+INT(RND*26))
160   b$=CHR$(48+INT(RND*10))
170   name$(i%)=a$+b$
180   h(i%)=i%
190   PRINT #ouch%, a$;b$
200   NEXT i%
210 REM -- random data just to test sort.
220 GOSUB 400 : REM sort them out
230 PRINT #ouch%
240 PRINT #ouch%, "For";n%;"items, order is:"
250 FOR i%=1 TO n%
260   PRINT #ouch%, name$(i%),h(i%)
270   NEXT i%
280 PRINT #ouch%
300 END
320 REM -- reader must modify to read & write file.
330 :
400 REM -- Shell-sort subroutine:
410 m% = n%
420 WHILE m% > 1
430   m% = INT(m% / 2)
440   IF m% MOD 2 = 0 THEN m%=m%-1
450   FOR i%=m%+1 TO n%
460     FOR j%=i% TO m%+1 STEP -m%
470       IF name$(j%)<name$(j%-m%) THEN GOSUB 800 ELSE j%=0
480       NEXT
490     NEXT i%
500   WEND
520 RETURN
550 :
800 REM -- swap routine:
810 r%=j%-m%
820 t=h(j%): h(j%)=h(r%): h(r%)=t
830 temp$=name$(j%)
840 name$(j%)=name$(r%)
850 name$(r%)=temp$
860 RETURN
870 REM uses t,temp$,r%,j%
880 :

A1
P8
C0
H5
R8
W9
C7
D4
```

```
Y2
T7
B3
E7
N6
B5
A1
P8
T4
L6
I5
I8
J4
D6

For 22 items, order is:
A1              1
A1              15
B3              11
B5              14
C0              3
C7              7
D4              8
D6              22
E7              12
H5              4
I5              19
I8              20
J4              21
L6              18
N6              13
P8              16
P8              2
R8              5
T4              17
T7              10
W9              6
Y2              9
```

Note that lines 120 to 200 were inserted merely to test the sorting procedure. You are asked to replace them with your own main program.

If you have time, you should incorporate the sorting module into the GOLFERS program (Listing 7.4) as an extra option.

6. One serious problem with the golf-handicapping program (Listing 7.4), even if the names are sorted, is that the user has to specify records for updating, deletion or display by number. On top of this, the user has to remember that record numbers go up two at a time.

It would be far easier if you could specify records by the name of the player concerned. Amend the record-selection routine, line 5000 forwards, so that the user gives a name, rather than a number, and the program uses BANKFIND to search through the RAM-file to find the number of the record containing that name. If no such name is found, it should ask the user to try again.

Now, after completing exercises 5 and 6, you actually have a useful program.

7. Write a bank account balancing program which maintains a file of cheques, standing orders, in-payments and other transfers so that at any time it can tell you your balance – or the size of your overdraft!

8

Advanced topics

THE PLOT THICKENS

There is a good deal more to the Amstrad range of computers than meets the eye at first glance. Our book is firstly about Basic programming, and only secondarily about the Amstrad, so we do not have space to do more than scratch the surface of its advanced features. Indeed the proper study of Amstrad Basic's sound and graphic facilities merits an entire book on each subject – and the titles of some are mentioned in the Bibliography.

However, we hope in this short chapter to explore some of the possibilities and stimulate some lines of future exploration.

8.1 Further graphics

There are three display modes on the Amstrad computer, selected by the MODE instruction. Each mode is characterized by its text and graphic resolution and its colour range. The screen resolution is the total number of individual points on the screen – called 'pixels' – that can be addressed. Text resolution refers to the number of rows and columns of characters that can be written to the screen. This table shows the attributes of each mode.

Mode	Graphic res.	Text res.	Colours
0	160 × 200	20 × 25	16
1	320 × 200	40 × 25	4
2	640 × 400	80 × 25	2

In all screen modes the graphics screen is addressed as though it has 400 vertical positions and 640 horizontal ones, but in modes 0 and 1 the resolution is not so fine – i.e. the lines are broader and the points are thicker.

There are 27 colours to choose from but a maximum of 16 (in mode 0) may be selected at one time. In mode 2, only two colours may be active. So you pay for high resolution with loss of colour. In fact mode 2 is best thought of as

the text-processing mode: you can fit 80 characters on a line for professional word processing and the like, but if you want pictures they have to be two-tone.

We have already begun to use some of the more primitive graphical facilities in earlier chapters. Let us recap on these basic capabilities before going deeper into Amstrad graphics.

We have used MODE to select screen-mode, purely to adjust the width of text lines. A MODE instruction has the side-effect of clearing the screen, but it is also possible to clear the screen with CLS, without changing mode.

Strictly speaking CLS clears the text screen only. There is a second screen, superimposed on the first, called the 'graphics screen' which can be cleared with the CLG instruction. So far we have not used the graphics screen at all.

We have, however, used the border instruction to surround the text screen with a contrasting colour. Its form is

BORDER colour1 [,colour2]

where 'colour' denotes an integer expression in the range 0 to 26. This fills the area around the screen with the colour given. The master list of colour numbers is printed on the CPC6128 itself. (See also the inside back flap of this book.) If the 'colour 2' parameter is present also, you get flashing colours: the two colours alternate. If you want to play with the rate of alternation, have a look at the SPEED INK instruction.

Finally we have used the LOCATE statement to achieve primitive 'block graphics' on the text screen – for instance, in Listing 6.6. The general form of this instruction is

LOCATE [#chan,] x,y

where 'chan' is a channel or window number and '**x**' and '**y**' give horizontal and vertical positions, measured from the top-left of the screen (which is position 1,1) in character units. Thus the **y**-coordinate is counted downwards: it refers to depth, not height. If the window specification is omitted, channel 0 is used, which is the whole screen.

8.1.1 Pen, ink and paper

We cannot go any further without explaining Amstrad's colouring scheme. As stated earlier, there are altogether 27 colours in the 'palette' – for example, by reference to the Master Colour Chart on your machine or on the back flap of the book, you will see that colour 12 is yellow, colour 22 is pastel green and so on. However, you can use at most 16 of these hues at any one time. In Amstrad terminology, you have 27 'colours' but only 16 'inks'. Going one step

further, in mode 1 (for example) you have only 4 different 'pens'. Let us see how the colour/ink/pen relationship works in practice.

Suppose we use MODE 1 to select a four-colour mode. Then by consulting the table of default ink settings (also on the back flap of this book) you will see that pen 3 is filled with ink 6 – i.e. bright red. (So, incidentally are pens 7, 11 and 15: the four colours are repeated four times over.)

But we can alter this state of affairs by a command like

INK 3,11

which puts ink 11 (sky-blue) into pen number 3. The INK command takes the general form

INK inkcode,huecode [,flashhue]

where 'inkcode' is an expression in the range 0 to 16 indicating one of the 16 notional pens, and 'huecode' is a colour number in the range 0 to 27 from the master colour chart. The optional 'flashhue' parameter is a second colour code in the range 0 to 27: the two colours will alternate if this is present, at a rate determined by the SPEED INK instruction.

Having filled our pens, we can put them to paper (well, glass actually) with two further instructions, PEN and PAPER.

The form of the PEN instruction is

PEN [#chan,] [inkcode] [,backmode]

where 'chan' is an optional window specifier. Either of the last two parameters may be omitted, but not both. The first, 'inkcode', selects the hue from the range 0 to 16, as assigned using INK (above) or by default (see back flap). The second governs whether the background will be transparent or opaque when drawing with this pen. Thus

PEN 3

selects the pen number 3 for the foreground colour until de-selected. Since no window is specified, this applies to the main screen display.

PEN selects the foreground colour. PAPER can be used to change the background colour. The form of the PAPER instruction is

PAPER [#chan,] inkcode

where 'chan' specifies a window, as above. If it is absent, as is the normal case, the main screen (window 0) is selected. The 'inkcode' is one of the inks, in the range 0 to 16, as set by INK or by default. So

```
100  MODE 0: PEN 0: INK 0,15
110  FOR p%=1 TO 15
120     PAPER p%
130     PRINT "Paper";p%
```

```
140     w$=INKEY$: IF w$="" THEN GOTO 140: REM wait
150     NEXT p%
200  END
```

runs through the spectrum of background colours, pausing after each one. The foreground colour, ink, zero, remains constant (orange, in fact).

All this discussion has applied, so far, to the text screen – i.e. to the foreground and background colours used for displaying text. There are two further instructions, however.

GRAPHICS PEN

and

GRAPHICS PAPER

which have the same format as the text versions PEN and PAPER (without the window-specifier options) but which apply to the graphics screen. Thus the foreground and background colours for drawing and plotting can be set independently from those for printing characters. E.g.

GRAPHICS PAPER 7

selects ink 7 (bright magenta by default) for the graphics background.

8.1.2 Lines and dots

To draw lines and shapes, we use three further instructions MOVE, DRAW and PLOT. This at last brings us into the realm of true graphics.

The MOVE statement has the form

MOVE x,y [,inkcode]

where '**x**' and '**y**' are the horizontal and vertical coordinates, while 'inkcode', if present, selects a new graphics foreground colour (range 0 to 16). The **x** and **y** axes of the graphics screen are measured from the bottom left, so this time **y** means height rather than depth (unlike the text screen where the vertical coordinate is measured downwards). The bottom left-hand corner of the screen is at position 0,0 and the top right is located at 640,400 in all modes. Thus

MOVE 320,200

positions the graphics pen in the centre of the screen; but it does not do any plotting.

To make a mark, you use the DRAW command, whose form is

DRAW x,y, [,inkcode]

which draws a straight line from the current **x/y** position to the **x/y** position

given in the instruction. Optionally, the 'inkcode' parameter may be used to alter the graphics foreground colour. For example

```
200 MOVE 0,400
220 DRAW 640,0, INT(RND*16)
```

moves the graphics pen to the top left of the screen (in line 200) and then draws a diagonal line to the bottom right-hand corner (in line 220). The colour of this line is randomly selected from the 16 inks; so if it just happens to be the same as the background colour, the line will be invisible.

The third of our genuine graphics commands is PLOT. This is used to plot a single point on the screen. Its form is

PLOT x,y [,inkcode]

where the parameters are as before. Its effect is to move to a particular **x/y** position on the screen and put a single point, or pixel, there in the graphics foreground colour. Thus

PLOT 320,200,2

switches on the point in the middle of the screen to the hue of ink number 2.

Listing 8.1 shows these graphics instructions in action.

Listing 8.1 Starburst

```
10 REM ******************************
11 REM ** Listing 8.1 :            **
12 REM ** STARBURST                **
15 REM ******************************
60 :
100 REM -- Random Star:
110 MODE 0: BORDER 16
120 CLS
130 RANDOMIZE TIME
140 star%=RND*24+4
150 FOR s%=1 TO star%
160   c%=INT(RND*16)
170   h%=INT(RND*640)
180   v%=INT(RND*400)
190   MOVE 320,200
210   DRAW h%,v%,c%
220   NEXT s%
300 END
```

This little program goes round a loop within which it selects a random colour, and random horizontal and vertical coordinates. Having chosen them, it MOVEs to the centre of the screen and DRAWs a line in the hue selected to the position given by **h%** and **v%**. The effect is to produce an explosion of coloured lines, like a firework. Try it.

The instructions DRAW, MOVE and PLOT are called 'absolute' plotting

instructions. They refer to points by addressing the 640 × 400 pixels on the screen. In contrast the related instructions DRAWR, MOVER and PLOTR are 'relative' plotting instructions. Instead of giving the absolute address of a point on the screen, they refer to points by their distance from the current position. For example,

```
200 DRAWR size%,0
220 DRAWR 0,size%
240 DRAWR -size%,0
250 DRAWR 0,-size%
```

draws a square box on the screen with the sides of length **size%** units. The exact position of this box is relative to the last point plotted.

8.1.3 Filling in the outlines

The instructions we have considered up till now can be used to draw outlines. The FILL command enables us to fill those outlines with colour, thus achieving solid shapes without the tedium of PLOTting hundreds of individual points. Its format is

FILL inkcode

where 'inkcode' is an expression in the range 0 to 16 for picking the hue. Its effect is to fill in, from the graphics cursor position, the area bounded by an enclosing shape. It fills till it reaches any boundary of the following three kinds: (1) the edge of the screen, or (2) a line in the graphics foreground colour, or (3) a line in the colour being used for filling.

Note that you must take care to move inside any outline to be filled, e.g. with MOVE; and if the shape is not closed the colour 'leaks out'.

The following example, demonstrates FILL in use. It draws a cluster of variously coloured balloons.

Listing 8.2 Coloured balloons

```
10 REM ******************************
11 REM ** Listing 8.2 :             **
12 REM ** COLOURED BALLOONS         **
15 REM ******************************
60 :
100 REM -- Cirles and ellipses:
110 MODE 1: BORDER 7
120 CLS
130 RANDOMIZE TIME
140 circles%=RND*16+4
144 turn=RND*2
150 FOR n%=1 TO circles%
160   c%=INT(RND*3)+1
170   h%=INT(RND*640)
180   v%=INT(RND*400)
```

```
190   warp=INT(RND*20)+25
200   GOSUB 1000 : REM draw oval
220   NEXT n%
300 END
330 :
1000 REM -- Circular Routine:
1010 MOVE 128,0 : REM bottom left
1020 up=v%: ac=h%
1030 DEG
1035 DRAW ac,up-warp,c%
1040 FOR circ%=0 TO 360 STEP 10
1050   up=warp*SIN(circ%): ac=44*turn*COS(circ%)
1055   IF circ%=0 THEN MOVE ac+h%,up+v%
1056   DRAW ac+h%,up+v%,c%
1070   NEXT circ%
1080 MOVE h%+1,v%+1: FILL INT(RND*4)
1100 RETURN
1110 :
```

The subroutine at line 1000 onwards is the balloon-maker. Line 1010 moves to the bottom of the screen, slightly off-centre to the left. Line 1035 draws the 'string' of the balloon in the colour selected in the main program. Lines 1040 to 1070 plot an approximately oval shape by a series of straight-line segments. Unlike some microcomputers, the Amstrad has no built-in circle or arc commands; so this is the only way to make curves. Finally, line 1080 fills the balloon just drawn with a randomly chosen colour. (Actually, because of the rules of FILLing, it may halt before it reaches the edge of the oval shape because it hits the edge or string of an overlapping balloon.)

Despite the program's simplicity, the resulting images are quite pleasing to the eye; and because of the element of randomness, they never quite repeat themselves. (Line 130, in case you are wondering, resets the random-number generator.)

The variables warp and turn determine whether the ovals will be short and fat or long and thin. If you want to know more, mug up a geometry textbook on ellipses.

One further command before you embark on your own image-making projects: ORIGIN. The normal graphics origin (zero point) is position 0,0 at the bottom left, as already stated. But by giving a command such as

ORIGIN xbottom,ybottom

you can reset it to the location specified by the two variables concerned. Subsequent plotting then takes place by reference to the new origin. (There are actually four other optional parameters to this instruction, for setting up a graphics window: see Appendices for more details.)

8.1.4. A picture emerges

There is a lot more to graphics on the Amstrad than we have room for here. You now have an 'explorer's kit' for this picturesque territory, but it is only fair to remind you what we have left out in the interests of brevity.

The following graphical keywords have not been covered in this section at all.

MASK
SPEED INK
SYMBOL, SYMBOL AFTER
TAG, TAGOFF
TEST, TESTR
XPOS, YPOS

The budding graphics programmer would be well advised to practise the fundamental instructions already presented, then look up some of the missing ones either in the Appendices to this book or in the User Instructions manual.

8.2 The sound of muzak

Sound output on the Amstrad is controlled by the SOUND instruction and modulated by two envelope commands (ENV for volume envelopes, ENT for tone envelopes).

8.2.1 The SOUND instruction

The SOUND instruction, at its simplest has the form

SOUND chan,tone [,duration [,loudness]]

where 'chan' selects the sound channel and its status, 'tone' selects the pitch of the sound, 'duration' specifies the time-period of the sound and 'loudness' gives the volume.

There are three channels, which can be controlled independently for three-part harmony. They are numbered 1, 2 and 4. The 'chan' parameter can take other values (for synchronization purposes), but we will leave that for now.

The tone is a numeric expression giving the reciprocal of the sound's frequency. This means that larger numbers denote lower tones. For example tone 239 is middle C at 262 cycles per second, while tone 142 is international A at 440 cycles per second – on a 'well-tempered' scale.

The duration parameter is specified in units of one-hundredth of a second.

Therefore a duration of 20 gives a sound lasting one-fifth of a second (0.2 seconds).

The loudness governs the amplitude of the sound, with 0 meaning off, 1 very quiet and 15 as the maximum volume.

Thus, the command

SOUND 1,144,40,12

causes the computer's speaker to beep on channel 1 at a pitch just off-key from international A for 0.4 seconds at 80% full blast. Try it. (You can alter the loudspeaker control manually with a little thumb-wheel behind the keyboard: maximum volume is really rather loud.)

And that is all you need to know to make music on the machine. The program below is a demonstration.

Listing 8.3 Pentatonic player

```
10 REM ******************************
11 REM ** Listing 8.3 :            **
12 REM ** PENTATONIC PLAYER        **
15 REM ******************************
50 :
100 REM -- The Pentatonic Scale & all that Jazz:
120 GOSUB 1000 : REM get notes
130 done%=64+INT(RND*64)
140 n%=note%/2
150 WHILE done%>0
170   loudness%=10+RND*5
175   duration%=beat%*RND+10
180   SOUND 1,tone(n%),duration%,loudness%
200   done%=done%-1
202   n%=n% + INT(RND*5) - 2
205   IF n%>note% THEN n%=0
210   IF n%<0 THEN n%=note%
212   PRINT n%
220   WEND
250 END
300 :
1000 REM -- Initialization routine:
1001 RESTORE
1010 beat%=12
1020 READ n%: note%=n%*5-1
1025 DIM tone(note%)
1050 FOR n%=0 TO note%
1060   READ tone(n%)
1080   NEXT
1100 RETURN
1110 :
1200 REM -- Musical data:
1202 DATA 5
1210 DATA 676,602,536,451,402
1220 DATA 338,301,268,225,201
1230 DATA 169,150,134,113,100
1240 DATA 84,75,67,56,50
1250 DATA 42,38,34,28,25
1300 :
```

The black keys (D-sharp, C-sharp, A-sharp, G-sharp and F-sharp) on the piano comprise a pentatonic scale. This has the supreme advantage of having almost no dissonances. It also sounds very ecclesiastical. The program above is a random pentatonic anthem generator. It is deceptively simple, but the resultant tone sequences (we hesitate to say 'music') are pleasant enough.

Subroutine 1000 initializes various parameters, including the array **tone()**. This holds five pentatonic scales whose notes are defined by the data statements on lines 1200 onwards. The lower notes come first. The main loop of the program from line 150 to 220 merely repeats from 64 to 127 notes. On line 170 the loudness is determined partly at random. On line 175 the duration is computed in a similar way. The SOUND statement on line 180 only uses channel 1: this is single-voice melody, without harmonies. The tone is selected from array **tone()** by the variable **n%**, which steps up or down at most two intervals each time round the loop. This increment, or decrement, is achieved on lines 202 to 210.

Line 212 is merely included to show you which note is being played – though you will notice that the screen display gets ahead of the sound output. This is because the sound-generator is an autonomous chip within the computer. The main CPU sends it a sound command and then can get on with something else. In fact, the sound-generator has its own queue of instructions, which it deals with in its own good time.

Rather than store the tone values in an array, as we do here, you may wish to calculate the appropriate tone value for each note. This can be done with the two lines

```
freq = 440 * (2 ∧ (Noct+((n−10)/12)))
tone = round(62500/freq)
```

where **Noct** is the octave number (−4 being the lowest and 3 the highest), **n** is the note of the 12-tone scale within that octave (1 being C, 2 being C-sharp, 3 being D etc.) and **freq** is the frequency in Hertz or cycles per second. Then tone is the value to use in the SOUND instruction.

8.2.2 What's inside the envelope

In its fullest form, the SOUND command can take as many as seven parameters (and as few as two). Only the channel and the tone are compulsory. Duration and loudness can be left out, in which case duration defaults to 20 (one-fifth of a second) and loudness to 12 (80% of maximum).

There are three further parameters – the volume envelope, the tone envelope and the noise selector, respectively. The last (noise) is a number from 0 to 31 which is best investigated experimentally: suffice it to say that it adds 'white noise' of various types to the output sound.

The envelope parameters are simply identifying numbers in the range 1 to 15. For them to have any effect, you must first define a volume envelope with ENV and/or a tone envelope with ENT.

The ENV instruction has an envelope identifier (1 to 15) then up to five sections each containing three parameters. That means it can have up to 16 parameters altogether. Each section has the form

n,s,p

where **n** is the number of steps in the section (0 to 127), **s** is the step size in volume levels (−128 to +127) and **p** is the pause time between steps (0 to 255, where 0 is treated as 256). The pause time is given in steps of 0.01 seconds, so that 2.56 seconds is the longest pause time.

Thus

```
110 ENV 2, 10,1,100, 40,-1,20
```

creates an envelope, number 2, with two sections. In the first section, lasting one second, the volume increases in ten steps (of 0.1 second each); in the second (lasting 0.2 seconds) the volume decreases in forty steps (of 0.005 seconds each).

Well, we never said it was going to be easy. Try sticking it in as line 110 of Listing 8.3 to see what it sounds like. The idea of a volume envelope is not hard to grasp: it specifies the rise and decay of the sound. But the details are very fiddly.

The ENT instruction works in an analogous manner. There is the envelope number (1 to 15) then one or more sections. Each section contains three values – the number of steps, the tone interval per step, and the time per step. You could try

```
112 ENT 2,100,2,2
```

in Listing 8.3. This, together with 110 above, would entail modifying line 180 of the program to

```
180 SOUND 1,tone(n%),duration%,loudness%,2,2
```

to use of the volume and tone envelopes.

For the musically minded, there is endless fascination here. For the rest of us, it is hard going. As far as putting things on paper is concerned, the best advice is: read the User Instructions, then experiment for yourself.

8.3 Interrupts

Charles Babbage, the grandfather of the modern computer, designed a blueprint for what he called an 'Analytical Engine' in the middle of the 19th

century. Apart from the fact that it was to be constructed from rods, cams, gears and so forth instead of electronic equipment, it was remarkably like the present-day digital computer. In fact, since Babbage's time, there have been only two conceptual advances in computer design of any significance.

The first was the idea of storing program instructions as data, usually credited to John Von Neumann, a mathematician who worked on the Manhattan project during World War Two. Babbage's machine treated programs and data in two different ways.

The second was the concept of the *interrupt*, which arose in the 1960s. An interrupt is a signal that interrupts normal processing and allows the computer to respond to events in the outside world. Without interrupts, computers could not control 'real-time' processes, nor could large mainframes share out time between many different users. Babbage's engine was not designed with 'multi-tasking' in mind.

It is rare for a microcomputer to provide interrupt-handling in Basic, but the Amstrad does just that. There are a number of keywords that enable a Basic program to interrupt what it is doing and attend to something else.

The AFTER instruction has the form

AFTER waittime [,timernum] GOSUB line

where 'waittime' is in fiftieths of a second, 'timernum' specifies which timer to use (0 to 3), and 'line' is the start of a subroutine for handling the event. After the given amount of time has elapsed, the subroutine will be executed. Each of the timers may have its own subroutine associated with it. Timer 3 has the highest priority, with timer 0 (the default) having the lowest.

The EVERY instruction has the form

EVERY timestep [,timernum] GOSUB line

where 'timestep' is again measured in fiftieths of a second; and the other parameters are the same as for AFTER. This calls the subroutine specified every so often. There is also a function

REMAIN(timernum)

that returns the amount of time remaining on the timer concerned (0 to 3) and disables it. These facilities allow the programmer to perform rather sophisticated time-slicing. For example, a loop awaiting user input such as

```
1200 ch$=INKEY$: IF ch$="" THEN GOTO 1200
```

could be interrupted from time to time, with a suitable message. The program need not get stuck waiting for something to be typed at the keyboard.

Note that you can turn off interrupts (apart from ESC) with DI (Disable Interrupts) if your interrupt-handling routine must not itself be interrupted by

anything else. Interrupts are re-enabled by EI or by the RETURN instruction that leaves the interrupt-handling routine.

Another similar facility allows trapping of errors and BREAKs. (A BREAK is caused when ESC is pressed twice.)

The

ON BREAK GOSUB line

instruction sets a trap so that whenever ESC is pressed twice, a particular routine is executed. The

ON BREAK CONT

disables the ESC altogether. After this command (until the next ON BREAK STOP) the ESC key will be ignored completely. Careful!

We have already used the

ON ERROR GOTO line

instruction (in Listing 6.6) to avoid halting the program when a missing file was specified for input. Once executed, this instruction tells Basic not to handle errors in the normal way, but to jump to a particular line instead. There the program can test what happened and act accordingly. To do so, use the **ERR** variable, which contains the error-number of the last error. You may also need to look at **ERL**, which holds the number of the line where the latest error occurred. A list of error numbers can be found in Appendix E.

After dealing with the condition that led to the error, the program can continue normally with the RESUME statement. If it fails to deal with the error, it can give up with

ON ERROR GOTO 0

which restores Basic's normal error-handling.

The net result of all these instructions is to enhance Basic considerably. They are there to make interactive video games easier to write in Basic, but they have other uses, as readers will no doubt discover for themselves.

8.4 Example program [RATMAZE]

Our example program illustrates one of the search strategies that is taught in introductory courses on Artificial Intelligence. It also provides a chance to show off a few more of the Amstrad's graphical facilities, especially the SYMBOL and SYMBOL AFTER instructions for defining new character-shapes.

The search process works with the concepts of 'open nodes' and 'closed nodes'. At any given time there may be several of both kinds in memory, each one on an incomplete pathway to the goal. A node is a single step along a route being constructed towards the goal. For each node, the computer

needs to store four pieces of information:

(1) where it is on the maze map;

(2) where it came from (its immediate predecessor) so the route can be re-traced later;

(3) how many steps have been taken to reach it;

(4) how far away it is from the goal.

These are held in the arrays **node()**, **path()**, **cost()** and **h()** respectively, which are dimensioned on line 88 of the program. They are used to represent the 'search tree' as it grows.

The search starts with only one open node: that is the location of the rat, who begins in the top left-hand corner. This initial node has no predecessor. The first step is for that node to be closed (so that the search will ignore it from then on) and all its successor nodes to be opened. These are the squares which can be reached with a single move (up, down, left or right) – except, of course any wall squares.

From then on the program grows new paths by picking the open node that looks most 'promising', then closing it and generating its successors as new open nodes – until it finds a solution or gives up. This cycle is the main loop at the heart of the program on lines 190 to 240.

On the screen, open nodes are marked by a ratlike symbol. This is printed after a PEN 15 instruction on line 4556 so that the rats appear in a flashing colour. Closed nodes (already examined) are shown as pawprints. The goal appears as a wedge of cheese. Blank squares have not been explored, and filled squares are the walls which block the rat's path.

The program does not use graphics proper, but it makes use of the Amstrad's facility for defining new character-shapes to give an interesting visual display. If you look at line 1550, for instance, where the food symbol (character 225) is defined, you will see how this works. The rather meaningless numbers are actually specifying the position of light and dark dots in an 8-by-8 grid. Each number defines a pattern of foreground and background colours on one of the eight lines. By staring at these numbers in binary notation, you may be able to perceive a deliciously cheesy shape. (Well, it looks all right on the screen!)

```
0     0 0 0 0 0 0 0 0
24    0 0 0 1 1 0 0 0
28    0 0 0 1 1 1 0 0
62    0 0 1 1 1 1 1 0
62    0 0 1 1 1 1 1 0
120   0 1 1 1 1 0 0 0
96    0 1 1 0 0 0 0 0
0     0 0 0 0 0 0 0 0
```

This method can be used to create new 8-by-8 character patterns of your own.

The progress of the search can be observed by watching the rat symbols marching outwards. Search strategies are usually taught as a rather dry subject; but with the aid of our robot rat, you can get an immediate pictorial grasp of what is going on. If there are several rats, it means that there are several partial paths being explored concurrently. At the end of the search, the maze is re-drawn and the route discovered marked out by footprints.

The main subroutines are as follows.

1000 Create the maze, partly at random so that it varies each time.
1500 Define the characters for rat, cheese, walls, and so forth.
2000 Clear the arrays that hold information about the open and closed nodes.
3000 Select the most promising node for further expansion.
4000 Expand the chosen node by closing it and opening its neighbours (successor nodes).
5000 Retrace the path discovered.

The significant variables are as follows.

mh Maze height
mw Maze width
path(I) Predecessor of node I
cost(I) Steps taken to get to node I
node(I) Position of node I in the maze
h(I) Heuristic estimate of distance from node I to goal

The heuristic estimate of distance from any node is calculated as the total number of squares (horizontally and vertically) to the goal. Thus the rat actually knows how far it is from the cheese, though it does not know about obstacles in its way till it hits them. The problem would be harder if the rat had a less accurate estimate of distance yet to travel (as happens in certain other tasks).

The variables **W1** and **W2**, set on lines 65 and 66, are also very important. By altering their values you can experiment with different search strategies. **W1** is the weighting given to the distance travelled so far (**cost**). **W2** is the weighting given to the heuristic estimate of distance still to travel (**hd**). They are used in subroutine 3000 to determine which open node is the most promising at each stage, and should therefore have its neighbours examined.

Initially **W1=1** and **W2=1.5**, but you can try other values to see their effect. A high weighting for **W1** is a bias towards conservatism. The system will then try to find the best possible pathway, but it will take longer (typically) to do so. A high value for **W2**, on the other hand, makes it biased towards

opportunism – only backtracking when forced to do so. This normally finds a way to the goal more quickly, but it is not always the optimum route. The best compromise is normally reckoned to be when **W1** and **W2** are both equal to **1**; but the search looks more interesting with higher values of **W2**, as you will see. (Try **W1=0** and **W2=1** for an extreme case.)

Listing 8.4 Rat-maze program

```
10 REM ************************
15 REM ** Listing 8.4 :      **
20 REM ** RAT-MAZE PROGRAM   **
30 REM ************************
40 MODE 0: BORDER 11
50 mh=18: mw=18: REM Maze Height & Width
55 size%=220 : REM tree limits
60 wa=1: rr=2: fo=3: dn=4: bl=5: fp=6
65 w1=1 : REM weight for SF
66 w2=1.5 : REM weight for HD
80 DIM m(mh+1,mw+1) : REM the maze
85 DIM c$(6) : REM maze characters
88 DIM path(size%),cost(size%),node(size%),h(size%)
90 REM Path, Steps, Node, Heuristic dist.
96 LOCATE 8,16: PRINT "Ratmaze!"
99 :
100 REM -- Rat in the Maze:
101 GOSUB 1500 : REM define graphics
110 GOSUB 1000 : REM make the maze
120 wmax=mw+1
150 nc=0 : REM no. of nodes examined
160 k=0 : REM counter
170 GOSUB 2000 : REM clear all paths
180 node(1)=2*wmax+2 : REM 1st open node
185 cost(1)=0: h(1)=fr-1 + (fc-1)
188 path(1)=0 : REM no predecessor
190 REM -- Main Loop:
195 WHILE (nc<size%) AND NOT (fr=sr AND fc=sc)
200   GOSUB 3000 : REM pick next node
210   nc=nc+1
215   LOCATE 1,mh+3
220   PRINT nc;SPC(2);sr;" ";sc;SPC(2);h(s);SPC(4)
230   GOSUB 4000 : REM generate successors
240   WEND
244 LOCATE 1,1: PRINT "Finished!!";CHR$(7);
250 IF fr=sr AND fc=sc THEN GOSUB 5000 : REM retrace steps
255 IF nc>=size% THEN PRINT " Failed!";CHR$(7)
256 PRINT nn;" nodes examined."
900 END
999 :
1000 REM -- Routine to create maze:
1001 PRINT" Wait a moment:": PRINT " making a maze."
1010 RANDOMIZE TIME
1050 FOR p=1 TO mh+1
1060   FOR r=1 TO mw+1
1070   IF RND < 0.33 THEN m(p,r)=wa ELSE m(p,r)=bl
1072   IF p=3 AND r<7 OR r=3 AND p<6 THEN m(p,r)=bl
1075   IF p=1 OR r=1 THEN m(p,r)=wa
```

```
1077   IF p>mh OR r>mw THEN m(p,r)=wa
1080   NEXT: NEXT
1090 fr=7 + INT(RND*(mh-7)) : REM food row
1100 fc=4 + INT(RND*(mw-4)) : REM food col
1110 m(fr,fc)=fo
1120 m(2,2)=rr : REM robot rat
1130 c$(bl)=" "
1140 c$(wa)=CHR$(227) : REM wall
1150 c$(dn)=" "
1160 c$(rr)=CHR$(224) : REM rat
1170 c$(fo)=CHR$(225) : REM cheese
1175 c$(fp)=CHR$(180) : REM footprint
1180 GOSUB 1200 : REM display maze
1190 RETURN
1199 :
1200 REM -- Maze display routine:
1210 CLS
1220 FOR r=1 TO mh+1
1230 FOR c=1 TO mw+1
1240 LOCATE c,r : PRINT c$(m(r,c));
1250 NEXT
1260 PRINT
1270 NEXT
1280 RETURN
1299 :
1500 REM -- Graphics definitions:
1510 PAPER 3
1515 PEN 1
1530 REM -- special characters:
1535 SYMBOL AFTER 220
1540 SYMBOL 224, 128,64,56,60,61,30,7,11
1550 SYMBOL 225, 0,24,28,62,62,120,96,0
1570 SYMBOL 227, 255,255,255,255,255,255,255,255
1590 RETURN
1599 :
2000 REM -- Tree-clearing routine:
2010 dd=9999 : REM marks unused cells
2020 FOR q%=1 TO size%
2030   path(q%)=0: cost(q%)=dd
2050   node(q%)=0: h(q%)=dd
2070   NEXT
2080 nn=1 : REM next free node
2090 RETURN
2099 :
3000 REM -- Pick best node S:
3010 s=1: bn=dd
3020 FOR i=1 TO nn
3025 sf=cost(i)
3030 hd=ABS(h(i))
3033 v=sf*w1 + hd*w2
3040 IF v<bn AND h(i)>=0 THEN s=i: bn=v
3050 NEXT
3060 IF s=1 THEN LOCATE 1,1:PRINT "Exploring:"
3065 IF s=1 THEN LOCATE 1,mh+2: PRINT "Steps Position  Dist"
3070 sr=INT(node(s)/wmax)
3080 sc=node(s) MOD wmax
3090 RETURN
3099 :
4000 REM -- Routine to generate successors:
4010 IF h(s)=0 THEN RETURN : REM quit early: solved.
4020 REM -- North:
4030 y=sr-1: x=sc
4040 IF y>1 THEN GOSUB 4400
```

```
4050 REM -- East:
4060 y=sr: x=sc+1
4070 IF x<=mw THEN GOSUB 4400
4080 REM -- South:
4090 y=sr+1: x=sc
4100 IF y<=mh THEN GOSUB 4400
4110 REM -- West:
4120 y=sr: x=sc-1
4130 IF x>1 THEN GOSUB 4400
4140 REM -- also close node s:
4150 h(s)=-h(s)
4160 IF h(s)>0 THEN PRINT "Help! Rat is stuck!";CHR$(7): STOP
4165 PEN 8
4170 LOCATE sc,sr: PRINT c$(fp);
4175 PEN 1
4180 m(sr,sc)=dn: REM blanks cell on screen
4190 RETURN
4199 :
4400 REM -- Routine to open one node:
4410 IF m(y,x)=wa THEN RETURN
4420 IF x>mw THEN RETURN
4425 xy=x + y*wmax
4430 REM -- find next free location:
4435 GOSUB 4800
4440 IF nx=0 THEN nn=nn+1 ELSE RETURN
4444 IF nn>size% THEN LOCATE 1,1: PRINT "I give up!";: STOP
4500 REM -- now open it:
4520 node(nn)=xy
4530 path(nn)=s
4540 cost(nn)=cost(s)+1
4550 h(nn)=ABS(y-fr) + ABS(x-fc)
4555 m(y,x)=dn : REM mark as visited.
4556 PEN 15
4560 LOCATE x,y: PRINT c$(rr);
4565 PEN 1
4570 REM shows it on screen.
4580 RETURN
4599 :
4800 REM -- Revisit check-routine:
4810 nx=0: IF m(y,x)<>dn THEN RETURN
4820 FOR n%=1 TO nn
4830   IF xy=node(n%) AND cost(n%)<=cost(s)+1 THEN nx=n%: n%=nn
4840   NEXT n%
4850 RETURN
4860 REM -- only reopen nodes if better path found.
4880 :
5000 REM -- Path-retracing routine:
5010 ts=cost(s)
5020 FOR qq=1 TO 2500: NEXT qq : REM delay
5025 GOSUB 1200
5030 LOCATE fc,fr: PRINT c$(fo);
5033 PEN 12
5040 WHILE s>0
5050   s=path(s) : REM parent node
5060   xy=node(s) : REM coords.
5070   x=xy MOD wmax
5080   y=INT(xy/wmax)
5090   m(y,x)=rr: REM rat's footprint
5100   IF xy>0 THEN LOCATE x,y: PRINT c$(fp);
5110   WEND
5115 PEN 8
5120 LOCATE 2,2: PRINT c$(rr)
5125 PEN 4
```

```
5130 LOCATE 1,mh+2: PRINT ts;" steps in path."
5140 PRINT nc;" nodes closed."
5150 RETURN
5160 :
```

The routine starting on line 4800 checks whether to re-open a previously visited node. It does so if the new way of getting there is quicker than the previous way. This can sometimes happen, and by taking advantage of such discoveries the program finds a route that appears more intelligent. (Note that when it draws an insoluble maze, as occasionally happens, the search fails gracefully.)

Try it and see. It is quite a clever little program, and visually appealing too. If you want to know more about search strategies in general, read *The Hitch-hiker's Guide to Artificial Intelligence* by Forsyth and Naylor (Amstrad edition from Chapman and Hall, 1986).

8.5 Exercises

1. Use the PLOT instruction in relative mode (or another method if you prefer) to create a subroutine that draws a regular polygon of specified side-length. The routine should also be given the number of sides and the position of the centre, plus a flag that tells it whether to fill the shape with colour or not.

2. Investigate the ENV and ENT commands further to try to find how to produce the sounds of snoring, chips frying, rain falling and heavy breathing. (They say it can be done.)

3. One of the Amstrad's disadvantages is that there is no simple way of dumping the contents of the screen to the printer (unless you exit to CP/M). We are not talking about a graphical screen-dump, merely about the sort of thing most microcomputers have whereby you press a special key and all that is typed (by user or machine) on the screen appears also on the printer – till further notice.

Write an interrupt-driven routine that gets called every second or so and tests whether the screen is nearly full. If it is, it should copy the text screen contents to the printer and then clear the screen. This should be written so that it can be MERGEd with any program to provide a hard-copy of its interactive screen input–output. (Very useful for example listings if you want to write a book called *The Amstrad Basic Idea*.)

4. Go one step further than the previous exercise and write a graphics screen-dump utility. This will involve you in looking into your printer's instruction manual. Most dot-matrix printers are capable of copying a screen

image: some can even do it in colour. Even daisy-wheel printers can manage it, after a fashion, though very slowly.

Because printers vary so widely, we can only offer very general guidance here; but you will find the XPOS and YPOS functions useful here. They will enable your program to scan the screen pixel-by-pixel in any order the printer requires.

9

Software design

STRUCTURED PROGRAMMING GUIDELINES

It is 2.00 a.m. You are all alone at the keyboard. You have been there, slogging away, since six-thirty in the evening. As you type RUN and press RETURN for the umpteenth time that night, you offer up a silent prayer that this time, at last, the program will work. You are now a fully paid-up member of the Midnight Hacker's Club.

How did you get into such a mess?

The short answer is through lack of planning. Once you reach the stage of praying that your programs will work, rather than knowing they will work, you have lost control. And once you lose control of software it turns into an insatiable Black Hole – swallowing up energy without visible result.

To help you avoid this Black Hole, we have devised a Midnight Hackers' rehabilitation plan. After all, we know the frustrations of midnight hacking ourselves.

9.1 Program design

There are three key questions to ask about any piece of software.

1) Does it work?

(2) Does it work?

(3) Does it work?

Most programs, unfortunately, fail all three tests. This includes many professional products.

However, after thirty-five years of software practice and experience, there is no excuse for this state of affairs – especially since it is easier to write software that works than software that does not.

Impatience is the enemy; and impatience in the form of deadlines (whether self-imposed or laid down from outside) undoes us all at times. So here are some guidelines which should help you remain in control, even under pressure.

9.1.1 Basic control structures

Over the years it has become accepted that there are four types of control structure which form the basis of a programmer's conceptual toolkit. These are listed below.

(1) Sequence

(2) Selection
    IF-THEN-ELSE

(3) Repetition
    WHILE-WEND
    FOR–NEXT

(4) Modularity
    Subroutines and Parameters

Sequence simply means doing one thing, then the next, then the next . . . and so on. That is how current computers work, so we hardly give it a second thought. The idea of analysing a problem into a sequence of steps takes a little getting used to, but most people pick up that knack quite quickly.

Selection means that a program can choose one path or another depending on conditions that arise during execution. This gives programs their great flexibility.

Repetition is what computers are good at. One of the arts of computer programming lies in formulating a solution as the repeated application of simple operations. We dealt with this in Chapter 3.

Modularity refers to the subdivision of a large system into components. It is not always obvious how a natural and efficient subdivision differs from an unnatural and/or inefficient one.

These four control structures are well-behaved and well-understood. Furthermore it has been proved that any computation can be expressed by a suitable combination of these, and only these, fundamental constructs. Amstrad Basic supports them all (with reservations, which we will discuss in Section 9.4).

Of course there are plenty of other control structures – including the notorious GOTO. But the point is that they (especially GOTO) give you exactly the sort of freedom you do NOT want, namely the freedom to get into a horrible mess.

This is the central message of structured programming: learn the four essential structures and how to put them together, and you have learned the fundamentals of programming in any language. Once you have mastered the basic constructs, you have the building blocks for a lifetime of productive programming.

The benefit of restricting yourself to these few standard forms is that you

stay in control, mainly because you never do anything particularly clever. This is because you have learnt that the smart way to produce a non-trivial piece of code is to put together lots of trivial pieces of code.

In other words, being a competent programmer is far more a matter of knowing how to use these structures than, say, remembering what **WAIT&FF84,20,25** means or that **INKEY(47)** indicates whether the user has hit the space bar. Such things can always be looked up.

9.1.2 Modularity

Of all the four fundamental programming constructs listed above, modularity is perhaps the hardest to master. Yet it is the one which repays understanding most handsomely.

A large system tends to be composed of several programs, and a large program usually consists of many routines. The routines in a program or programs in a system are termed 'modules'.

The objective of modular programming is to parcel up the program into thought-sized chunks, each with a well-defined purpose. In addition, the chunks should communicate with one another in a tidy fashion. How can we achieve these objectives?

Generally speaking, you are safer with smaller modules, for two reasons. Firstly, a small module is easier to code, easier to test and easier to understand than a large one. Secondly, a module that does only one job is more likely to be generally useful, for instance in another program or another part of the same program. In software, as in economics, small is beautiful.

It is wise, therefore, to avoid the temptation to add extra features to your procedures. Concentrate instead on making them do one thing well.

For example, it might seem a good idea to write a procedure

```
GOSUB 3600 : REM sorts and prints array
```

in preference to

```
GOSUB 4000     : REM sorts array into order
GOSUB 4400     : REM prints it out
```

but one day you may want sorting without printout, or sorting followed by output in different format. And in any case it is easier to test one thing at a time. If the sort-and-print routine fails to work properly, you have to discover whether the sorting part or the printing section is going wrong. If one of the others fails, you know where to start looking.

Having chopped your big program into small modules which are implemented as routines (GOSUB and RETURN) or functions (DEF FN), you must ensure that the modules pass information among themselves in a well

regulated manner. Amstrad Basic makes it difficult to pass information into a subroutine via parameters (see Chapter 4) because all variables are global to the whole program. Global variables are a very convenient means of communicating among subroutines, but the trouble with global variables is that one routine can make changes that inadvertently affect another. This can cause chaos. We shall have more to say on this question in Section 9.4.

9.1.3 Stepwise refinement

Granted that we want modules that are small, simple and good at one specific job, how do we turn a blank piece of paper into a collection of efficient procedures and functions? (Yes, good old-fashioned paper! You do not start typing a program in until you have written down what you are going to type.)

One answer to this problem is known as Stepwise Refinement, or Stepwise Decomposition. It is a top-down approach to problem analysis.

First you split the main program into major processes. Then you break the major processes into a small number of subdivisions. Each of these is given a name and the task that it accomplishes is defined by stating what inputs it requires and what outputs it delivers. How the subprocess actually carries out its job is not relevant at this stage.

Once a module is coded in this way, attention is focused on each of its subprocesses in turn, using the same strategy of decomposition. Ultimately subprocesses are reached which are trivial. They can be coded directly so the decomposition halts.

Although it appears that all the complexities are relegated at every stage to lower levels, it will be found in the end that nothing complicated remains to be done. That, at least, is the theory!

We shall attempt to put this theory into practice in Chapter 10, so you can judge us by results, but what we are saying – in a nutshell – is that writing a well-structured program is rather like writing a well-organized book.

You start with an outline, which is little more than a list of chapters and topics. You push these around for a while, considering alternative arrangements and different ways of subdividing the topics you want to cover. When you are happy with the overall plan, you take each chapter in turn and split it up into sections under a number of different subheadings. Under each subheading you write a few paragraphs, each composed of a small number of sentences concerned with a single theme.

As long as the chapters follow a logical plan, and the sections within each chapter follow a logical sequence, and the paragraphs in each section also follow a logical sequence, there is a good chance that the whole structure makes sense.

9.1.4 An implementation timetable

Programs do not spring ready-made from the programmer's head. They evolve over time.

The growth (and decay) of software with time is an issue that is very often ignored. But the time dimension is a crucial aspect of software design.

Many over-eager novices come to grief by trying to get a complete system, with all the 'bells and whistles', working from the beginning. So do quite a number of old hands who ought to know better. It is wiser to adopt an incremental approach, biting off only one mindful to chew it over at one time.

This entails a timetable or plan of campaign – something along the lines shown below.

Days 1–2:	Write the skeleton main program with only the main menu procedure fully working.
Days 3–5:	Plug in and test the input routines.
Days 6–7:	Get the output modules working.
Days 8–10:	Code and test the updating functions.

Even an outline as simple as this will prove helpful for the amateur programmer, though a professional would want something more detailed.

The advantage of getting an initial skeletal version working early on is that it gives you encouragement. It also lets you see the consequences of your design decisions while there is still time to re-think them.

If you are writing a program for others to use, you can show them the prototype and obtain their comments. It is vital to receive this kind of feedback at an early stage before your ideas are 'set in concrete'. We all know how hard it is to change our minds once we have become accustomed to a certain way of thinking.

The life-cycle of a typical program looks something like this.

```
REM planning
REM outline
REM prototyping
REM testing
WHILE NOT obsolete
   REM modification
WEND
END
```

If you think the job is done when version 1.0 finally runs to completion you have either misunderstood the software life-cycle or written a very short-lived program. Most programs are subjected to continual revision. (One hesitates to say 'improvement'!) This is because neither the system's designers nor its users really know what they want until they see what they have got.

For this reason, flexibility is almost worth more than you are prepared to admit at the moment of coding. Let us give a small example.

If you ever write an output procedure that works perfectly on 66-line printer paper (and only on 66-line printer paper) you will surely live to regret it. It takes a little extra time to write it so that it can be switched to 60-line or 72-line paper – merely by setting an extra variable giving the page length – but not nearly as much time as it will take to alter all the references to 66 (and perhaps 65 and 67) in the code, and to check you have got all the changes right. Make no mistake: one day your stockist will run out of the right sized listing paper, or you will buy a new printer. One of the cardinal rules of the software game is that the rules keep changing. You cannot even play, let alone win, unless you plan for change.

In our pseudo-program above, the system gets modified until it is obsolete. Some people will find it strange that a program with no moving parts and nothing to wear out or go rusty could eventually break down; but it does happen. It happens because the world moves on, people develop new requirements, the hardware becomes out of date, and so forth.

By emphasizing the temporal development of software in its human context we are encouraging you to design systems with a long and useful life ahead of them. Lack of modifiability leads to the premature senility of software.

A program in its old age is a sorry sight – abandoned long ago by those who cared about it and so full of scars, stitches and emergency transplants that all trace of its original form is obliterated. No one understands any longer how it works or, more important, why it goes wrong. It becomes a chronically sick patient, languishing in the geriatric ward, puzzling its doctors and annoying the nurses by its erratic behaviour and frequent demands for attention. One day the time comes when it is less trouble to put it out of its misery than to attempt the drastic surgery needed to restore it to its former health.

9.2 Data representation

It would be quite wrong to give the impression that program structure is the only thing that matters in software design. Data structuring is even more crucial. And a mismatch between the structure of the program and the structure of the data it uses is likely to result in a program that is grossly inefficient, if indeed it works at all.

Yet many programmers still regard data representation as an optional extra – to be dealt with as an afterthought when the real business of coding is complete. This attitude is mistaken. Most non-trivial programs have to replicate reality in some sense, and the accuracy of their world model

determines how well they work. This comment does not apply only to explicit simulation programs: a weather forecasting system models the atmosphere; a stock control program models the behaviour of customers; a game-playing program models, in a sense, the human planning process; and so on.

9.2.1 Data structuring

Basic is not very well endowed with data structuring facilities. The trick is to learn how to build on what is provided.

The simple data types – characters, integers and floating-point or 'real' numbers – are the materials from which all data representations have to be built. The only structured data objects provided in Basic are the array and the string.

Datafiles are different in that they can contain relatively large amounts of data and are slower to access; but Amstrad Basic treats files like very long strings of bytes. As we saw in Chapter 7, any structure the programmer wants has, in effect, to be imposed on the file.

We have dealt with the conventional uses of arrays in Chapter 3 and of strings in Chapter 5. Here we want to consider how to use them to create data structures that are not part of the Basic language. We shall consider examples that mimic three or more complex structures – tables, trees and networks.

9.2.2 A table of records

A table is an array of records. It is rather like a datafile which is small enough to be held in main memory (RAM) and thus need not reside on disc. Each record describes one object or entity and is composed of several fields.

To be specific, let us imagine that we are starting to design a program which will work out routes on the London Underground and use the Amstrad's excellent colour graphics to display its results as a map on the screen. (Later it could be generalized to the Paris Metro or the New York Subway, provided we keep it flexible.)

In this example, the objects we are interested in representing are stations, and the lines that connect them. A station record, for the time being, will have the following four fields.

Fieldname	Type	Description
name$	string	The station's name
tubeline$	string	The line it is on
high%	integer	The **y**-coordinate on the screen map
wide%	integer	The **x**-coordinate on the screen map

e.g.

name$	"Finsbury Park"
tubeline$	"Victoria"
high%	106
wide%	144

A table of such objects is simply a one-dimensional array of station records. But Basic does not provide a way of grouping the four different items into a single unit, so we have to link them together ourselves.

With the statements

```
100  stations% = 100
110  DIM name$(stations%), tubeline$(stations%)
120  DIM high%(stations%), wide%(stations%)
```

we can declare enough storage for a table of 100 stations.

Notice that by employing the integer arrays **high%** and **wide%**, we are assuming that the **x** and **y** coordinates of our underground stations have no fractional part. In practice we want then to lie in the range 0 to 400 in **high%()** and 0 to 640 in **wide%()**.

Now at last we are ready to read in a table of stations.

```
1000  RESTORE
1010  FOR S% = 1 TO stations%
1020  READ name$(S%), tubeline$(S%)
1030  READ x%, y%
1035  wide%(S%) = x% : high%(S%) = y%
1040  NEXT S%
....
2000  DATA Newbury Park, Central, 186, 107
2010  DATA Barkingside, Central, 186, 110
2020  DATA Fairlop, Central, 186, 114
2030  DATA Hainault, Central, 186, 118
2040  DATA Liverpool Street, Circle, 150, 75
2050  DATA Moorgate, Circle, 140, 82
2060  DATA Barbican, Circle, 136, 82
....  [ and so on ] ....
```

It is now time to introduce the notion of 'data hiding'. The idea is that we hide the details of how we implement a data structure behind a series of subroutines and functions, which present the data the way we want it, not the way the machine handles it.

As you will have noticed, we created a set of four arrays not a single table of records with four fields. It is only the way we use those arrays that links them together. In other words, we always use the same subscript for related data. Thus

```
name$(25)
```

contains the name of the 25th station, and

```
high%(25)
```

contains the vertical screen position for the 25th station.

But this only remains true as long as the programmer is consistent. By hiding away the arbitrary details, we can make it less easy for mistakes to creep in. For instance, it makes sense to define a few routines that handle our four-part items as wholes.

The following two routines plot a station's name at the correct place on the screen (subroutine 3000) and compute the distance between two stations (FNdist) using Pythagoras's Theorem.

```
3000  REM uses s% as a parameter:
3020  REM displays name at its screen location:
3030  MOVE wide%(s%)*4, high%(s%) * 4
3040  TAGON      : REM write at graphics cursor
3050  PRINT name$(s%);
3060  TAGOFF     : REM write at text cursor
3070  RETURN
3200  DEF FNdist(s1%,s2%)=SQR((high%(s1%)-high%(s2%))^2 +
      (wide%(s1%)-wide%(s2%))^2)
```

The point of this is to be able to refer to what we think of as a single thing (a station) by one data item, e.g.

```
s% = stat%: GOSUB 3000
```

to plot a given station. We do not want to keep being forced to remember that the computer is in fact treating each item as four separate things.

A side benefit comes, as we shall see, when we alter the data format so that each station has more than four pieces of information associated with it. We may have to alter some of the routines, but we do not have to comb through an entire program searching for references that may be incorrect, because we can still refer to a single entity by a single datum – its station-number.

9.2.3 A tree structure

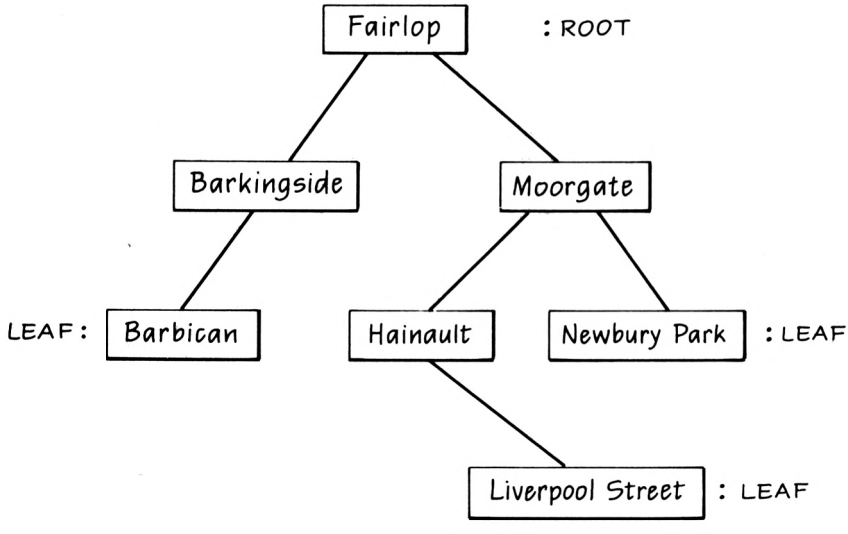
Trees and networks are flexible data structures that computer scientists have found useful for representing relationships between data elements. Neither is part of Basic, but we will pursue our station-data illustration a bit further to demonstrate that such things can be implemented in Amstrad Basic.

One rather efficient method of sorting data, and of accessing it, involves building up what is called an ordered Binary Tree.

A binary tree contains 'nodes', each of which contains some data. In

addition each node has a left pointer and a right pointer. The left pointer points to its left-hand branch and the right pointer to the right-hand branch. Either one may be absent. If both are absent the node is called a 'leaf' node. There is one special node, the 'root': it is not on any other node's left or right branch. Figure 9.1 illustrates this terminology in use.

Fig. 9.1 An ordered binary tree



In the diagram, the root node is Fairlop. Barbican, Liverpool Street and Newbury Park are leaf nodes. Moorgate has both left and right branches, while Barkingside has only a left branch and Hainault only a right branch.

This tree has one other important property. From any node, if you follow down from its left branch, you will only find names which come before it in alphabetic order, if you follow its right branch, you will find only names that come after it in alphabetic order.

By following simple rules we can build up a tree of this kind given data in a haphazard order. Once we have such a tree we can quickly find whether a particular name is in it. This is why binary trees are often used as indexes. We can also easily print out the data in the correct order. It is the latter task that we illustrate here with an example.

The following program builds up and displays an ordered binary tree, using our station data, thereby accomplishing a sort of their names.

Listing 9.1 Binary tree creation

```
10 REM ******************************
11 REM ** Listing 9.1 :            **
12 REM ** BINARY TREE CREATION     **
15 REM ******************************
60 :
100 REM -- Binary-tree for Station Data:
101 ouch%=8 : REM 8 for printer
110 MODE 2: BORDER 16
120 RESTORE
130 READ stations%
140 root% = 1
150 DIM name$(stations%), tubeline$(stations%)
160 DIM high%(stations%), wide%(stations%)
170 DIM lh%(stations%), rh%(stations%)
175 REM data arrays for tree structure.
180 DIM t%(20) : REM stack for routine 1100
200 k%=0 : REM counter
202 d%=0 : REM initial depth counter
210 n% = stations%: GOSUB 1000 : REM read tree
220 t%(1)=root% : GOSUB 1100 : REM show tree
240 PRINT #ouch%
250 GOSUB 2000 : REM dump out tree.
300 END
330 :
1000 REM -- Read-tree routine:
1010 REM parameters: n%, s%:
1020 FOR s%=1 TO n%
1030   GOSUB 1500 : REM read one station
1040   NEXT s%
1050 RETURN
1060 :
1100 REM -- Show-tree procedure:
1110 REM t%() holds parameters:
1120 REM uses recursion.
1130 d%=d%+1 : REM one more level down.
1140 IF t%(d%) < 1 THEN d%=d%-1: RETURN
1150 t%(d%+1) = lh%(t%(d%)) : REM plant left subtree
1160 GOSUB 1100 : REM calls itself
1170 ss%=t%(d%) : REM this node%
1180 GOSUB 1300 : REM dsiplay station.
1190 t%(d%+1) = rh%(t%(d%)) : REM plant right subtree
1200 GOSUB 1100 : REM recursive call
1210 REM does lh, then self, then rh.
1220 d%=d%-1 : REM prepare to rise
1230 RETURN
1250 :
1300 REM -- Show-station routine:
1310 REM ss% is input.
1320 k%=k%+1
1330 PRINT #ouch%, k%;TAB(7);name$(ss%);"  ";tubeline$(ss%);
1340 PRINT #ouch%, TAB(33);high%(ss%);" ";wide%(ss%)
1350 RETURN
1360 :
1500 REM -- Read-station routine:
1510 REM s% is input:
1530 READ name$(s%), tubeline$(s%)
1540 READ wide%(s%), high%(s%)
1550 lh%(s%)=0: rh%(s%)=0
1560 ns%=s% : GOSUB 1650 : REM link station into tree.
```

```
1570 REM insert into growing tree.
1580 RETURN
1590 :
1650 REM -- Link-station routine:
1660 REM ns% is input; uses next%, r%:
1670 IF ns%=1 THEN RETURN : REM 1st one.
1680 r%=1: newnode%=1
1690 WHILE newnode% <> 0
1700   IF name$(ns%) <= name$(r%) THEN newnode%=lh%(r%)
1710   REM if lesser look down left branch.
1720   IF name$(ns%) > name$(r%) THEN newnode%=rh%(r%)
1730   REM if greater look down right branch.
1740   IF newnode%>0 THEN r%=newnode%
1750   WEND
1760 REM has now found where to put item ns%
1770 IF name$(ns%) > name$(r%) THEN rh%(r%)=ns% ELSE lh%(r%)=ns%
1780 RETURN
1790 :
2000 REM -- Station-dump routine:
2010 REM uses k% as input:
2020 PRINT #ouch%
2030 n% = k% / 2 : REM roughly half.
2040 FOR i% = 1 TO n%
2050   PRINT #ouch%, i%;TAB(7);name$(i%);TAB(25);
2060   PRINT #ouch%, USING "##  ##"; lh%(i%),rh%(i%)
2070   NEXT i%
2080 RETURN
2090 :
9000 REM -- Specimen data (just a small selection):
9010 DATA 28
9020 DATA Queens Park, Bakerloo, 65, 95
9030 DATA Kilburn Park, Bakerloo, 67, 93
9040 DATA Maida Vale, Bakerloo, 69, 91
9050 DATA Warwick Avenue, Bakerloo, 71, 89
9060 DATA Paddington, Bakerloo, 75, 88
9070 DATA Edgware Road, Bakerloo, 82, 88
9080 DATA Marylebone, Bakerloo, 89, 88
9090 DATA Baker Street, Bakerloo, 96, 88
9100 DATA Regents Park, Bakerloo, 105, 82
9110 DATA Oxford Circus, Bakerloo, 105, 75
9120 DATA Piccadilly Circus, Bakerloo, 110, 67
9130 DATA Charing Cross, Bakerloo, 115, 62
9140 DATA Embankment, Bakerloo, 116, 55
9150 DATA Waterloo, Bakerloo, 116, 48
9160 DATA Lambeth North, Bakerloo, 119, 44
9170 DATA Elephant & Castle, Bakerloo, 122, 40
9180 DATA Leytonstone, Central, 169, 98
9190 DATA Leyton, Central, 169, 91
9200 DATA Stratford, Central, 169, 84
9210 DATA Mile End, Central, 167, 76
9220 DATA Bethnal Green, Central, 161, 75
9230 DATA Liverpool Street, Central, 151, 75
9240 DATA Bank, Central, 140, 75
9250 DATA St Pauls, Central, 136, 75
9260 DATA Chancery Lane, Central, 132, 75
9270 DATA Holborn, Central, 122, 75
9280 DATA Tottenham Court Road, Central, 115, 75
9290 DATA Oxford Circus, Central, 105, 75
9300 REM -- plenty more where that came rom!

1     Baker Street  Bakerloo     88   96
2     Bank  Central               75   140
3     Bethnal Green  Central     75   161
```

```
4     Chancery Lane  Central      75    132
5     Charing Cross  Bakerloo     62    115
6     Edgware Road  Bakerloo      88    82
7     Elephant & Castle  Bakerloo
                                  40    122
8     Embankment  Bakerloo        55    116
9     Holborn  Central            75    122
10    Kilburn Park  Bakerloo      93    67
11    Lambeth North  Bakerloo     44    119
12    Leyton  Central             91    169
13    Leytonstone  Central        98    169
14    Liverpool Street  Central   75    151
15    Maida Vale  Bakerloo        91    69
16    Marylebone  Bakerloo        88    89
17    Mile End  Central           76    167
18    Oxford Circus  Central      75    105
19    Oxford Circus  Bakerloo     75    105
20    Paddington  Bakerloo        88    75
21    Piccadilly Circus  Bakerloo
                                  67    110
22    Queens Park  Bakerloo       95    65
23    Regents Park  Bakerloo      82    105
24    St Pauls  Central           75    136
25    Stratford  Central          84    169
26    Tottenham Court Road  Central
                                  75    115
27    Warwick Avenue  Bakerloo    89    71
28    Waterloo  Bakerloo          48    116


1     Queens Park          2    4
2     Kilburn Park         6    3
3     Maida Vale          15    5
4     Warwick Avenue       9   14
5     Paddington           7   11
6     Edgware Road         8   13
7     Marylebone           0   10
8     Baker Street         0   12
9     Regents Park         0   19
10    Oxford Circus       20    0
11    Piccadilly Circus    0    0
12    Charing Cross       21    0
13    Embankment          16   26
14    Waterloo             0    0
```

To clarify what is going on, we have included a routine at line 2000 to display the data as stored. Let us inspect the data as it is after four stations (Queens Park, Kilburn Park, Maida Vale and Warwick Avenue) have been added, in that order. We ignore the **x** and **y** coordinates, just to keep things simple.

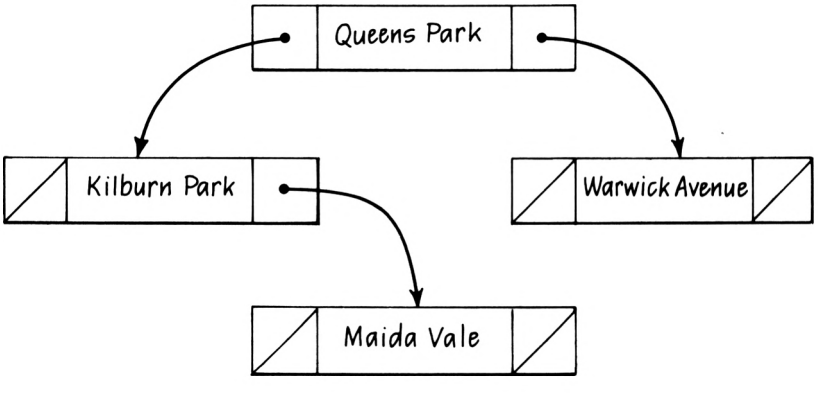
	name$	**line$**	**lh%**	**rh%**
1.	Queens Park	Bakerloo	2	4
2.	Kilburn Park	Bakerloo	0	3
3.	Maida Vale	Bakerloo	0	0
4.	Warwick Avenue	Bakerloo	0	0

Queens Park is at position 1, the root. It has two subtrees, the left one can be found at position 2 (**lh% = 2**) and the right one at position 4 (**rh% = 4**)). At position 2 is Kilburn Park. This has a right branch (**rh%=3**) but no left branch (**lh%=0**). (Later it will have number 6, Edgware Road, on its left branch as shown by the sample output.) At position 3 is Maida Vale, and at position 4 is Warwick Avenue. Neither has left or right branches yet, so they are leaf nodes, though they will acquire subtrees of their own as new data arrive.

As new nodes are inserted they are threaded into the tree by subroutine 1650, but Queens Park remains at the root. All that changes is that some of the zeros in **lh%** and **rh%** are replaced by addresses of newly added stations.

Thus **lh%** and **rh%** are being used as **pointers**. They give the address of the place where the left and right subtrees can be found, or zero to indicate the absence of a subtree. We can depict the situation as follows.

Fig. 9.2 Tree diagram



The arrows linking this structure together have been implemented by integers which point to the correct place in the arrays. By following the pointers the program can traverse the structure. Thus we have turned our table into a tree, still using only arrays and strings.

Notice that the routine at line 1100 is recursive. It calls itself. This makes it relatively simple to write and, once you get used to recursion, to understand. The idea behind it is simple. If the present node has a left subtree, that contains names to be printed before it. How better to do so than by calling the procedure itself? Then the data at the current node is printed. Finally, if it has a right subtree, that is printed too – using subroutine 1100, of course. (See also Chapter 4.)

Here the routine to process a structure works well recursively because

the structure itself is recursive. A tree contains subtrees, the left and right branches, which are themselves trees. (This is an example of what we mean by matching the program structure to the data structure.) As long as the tree is built up correctly by subroutine 1650, it will be displayed in the correct order by subroutine 1100.

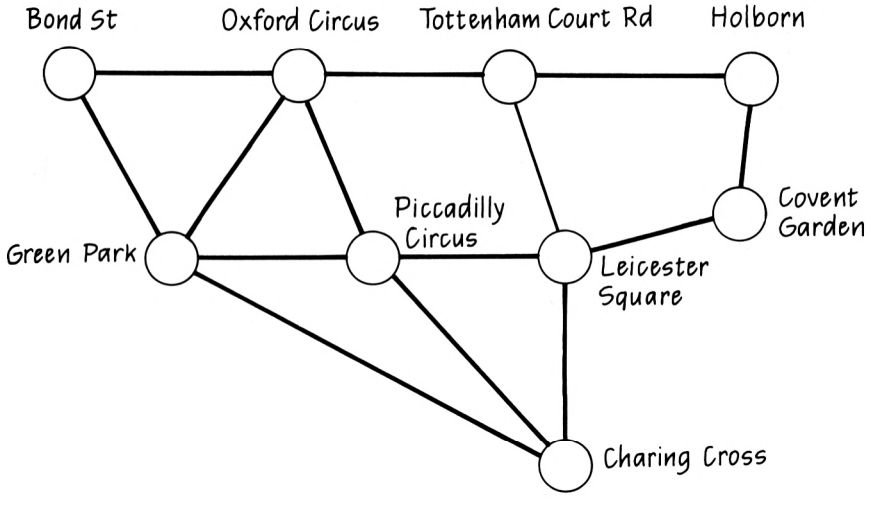
Data implementation details are comparatively well hidden, since each procedure works with a single integer that refers to all the bits and pieces that go together as a unit. However we had to introduce a stack, **t%()**, to achieve the recursion in subroutine 1100. The stack is another useful data structuring concept, which in this case we implemented using an array. This is necessary because each 'level' of subroutine 1100 needs its own copy of the tree or subtree to be listed. This is held in **t%(d%)** with **d%** augmented as the recursion descends and decremented as it ascends. (This idea is discussed further in Section 9.4.2.)

9.2.4 A network

A network is one step further in complexity than a tree. Each node in a network can have more than two links, and there is no special root node. There may indeed be loops in a network. The Underground system is very naturally represented as a network.

The following shows a small portion of the London Underground as a network.

Fig. 9.3 Network diagram (London Underground)



Since there are only nine stations here, it is possible to represent the connections in the network by a two-dimensional array.

	1.	2.	3.	4.	5.	6.	7.	8.	9.
1. Bond Street	1	1	0	0	1	0	0	0	0
2. Oxford Circus	1	1	1	0	1	1	0	0	0
3. Tottenham Court Rd	0	1	1	1	0	0	1	0	0
4. Holborn	0	0	1	1	0	0	0	1	0
5. Green Park	1	1	0	0	1	1	0	0	0
6. Piccadilly Circus	0	1	0	0	1	1	1	0	1
7. Leicester Square	0	0	1	0	0	1	1	1	1
8. Covent Garden	0	0	0	1	0	0	1	1	0
9. Charing Cross	0	0	0	0	1	1	1	0	1

We place 1 in a cell to indicate that two stations are connected, and zero otherwise. (We also made the assumption that each station is linked to itself.) Thus there is a 1 at position [6,5] indicating that station number 6 (Piccadilly Circus) is linked directly to number 5 (Green Park). At position [4,7] there is zero, showing that Holborn is not connected with Leicester Square.

If necessary we could go further and insert not simply ones and noughts, but numbers giving the time taken to go from one to another (with a negative value indicating the absence of a direct link) – provided we had that information.

But this is actually a very poor choice of data representation. The trouble is that there are 271 stations in the actual network. To link them in this way would need a matrix of 271 * 271 = 73441 cells. Most of those cells would be empty since no station has more than seven neighbours.

We can represent this network in a far more economical fashion using three separate one-dimensional arrays. Assuming that the average number of neighbours is not more than four per station, we can declare

DIM name$(stations%), link%(stations%), list%(stations%*4)

where **name$()** is for station names as before, **link%()** is for a pointer to a list of neighbours of each station and **list%()** contains the actual positions of those neighbours. The connections among the first four of the stations shown above in the matrix could be represented as below.

name$	**link%**	[refers to **list%**]
1. Bond Street	1	→ **list%(1)**
2. Oxford Circus	3	→ **list%(3)**
3. Tottenham Court Rd	6	→ **list%(6)**
4. Holborn	9	→ **list%(9)**
5.		

	list%	[refers back to **name$**]	
1.	2	→ **name$(2)**	[Oxford Circus]
2.	0		
3.	1	→ **name$(1)**	
4.	3	→ **name$(3)**	
5.	0		
6.	2	→ **name$(2)**	
7.	4	→ **name$(4)**	
8.	0		
9.	3	→ **name$(3)**	
10.	0		
11.	....		

Here we are using a value of zero in the **list%()** array to mark the end of a chain of neighbouring stations. We have also put an explanation of the stations linked in brackets after the non-zero entries to help you find your way around. The essential point is that numbers in the **link%()** array refer to the **list%()** array, while numbers in **list%()** refer back to the **name$()** array.

The routine that follows displays the neighbours of any given station, showing how this structure might be put to use.

```
2500 REM parameter s%
2505 REM ------------
2510 PRINT "From Station ";name$(s%);" you can reach :"
2520 s% = link%(s%)
2525 WHILE list%(s%) <> 0
2530     PRINT TAB(13);name$(list%(s%))
2540     s% = s% + 1
2550     WEND
2555 RETURN
```

With less than ten nodes in our network, the additional work of doing this kind of linkage is not worthwhile. But with more than a few dozen nodes, it could make the difference between a feasible and an impossible system.

To sum this section up, we have tried to make two important points. Firstly, you can remedy some of Basic's weaknesses in data organization by using integer pointers to link up structures of your own devising. Secondly, you are well advised – if you attempt to build more complicated structures than Basic provides – to hide the fiddly details within particular routines. Then each routine performs a simple operation on data elements that you have defined, and you can think in terms of operations on structures appropriate to the problem rather than structures appropriate to the machine.

9.3 When things go wrong

All right. You claim to have followed all that advice, and your program still does not work. If you hear a grinding sound it is the gnashing of the author's teeth.

In a way you are lucky. At least you know that something is wrong. There are plenty of programs around whose users are blithely unaware they are getting incorrect results.

We can start from the assumption that you have found some sort of mistake or malfunction – a 'bug' in computer parlance. It is helpful to divide program bugs into three categories: (1) syntax errors; (2) execution errors; and (3) logical errors. Each kind is more serious than the one before. We will deal with them in turn.

9.3.1 Syntax errors

These occur when the program violates the rules of Basic. Amstrad Basic does not detect errors of syntax when a line is typed in, so you have to wait till the line is executed for the system to complain that it cannot understand your language, and halt the program.

Syntax errors are normally the easiest kind to deal with. In many cases they arise from simple misspellings, typing mistakes, missing commas and so forth. Since the Basic interpreter tells you where the error occurred, it is usually just a matter of re-typing PRONT as PRINT or inserting a missing quotation mark or something like that.

Occasionally syntax errors are symptoms of a deeper problem. What typically happens in such cases is that when the syntax errors are corrected, the program still fails to run.

9.3.2 Execution errors

Anyone who has taught programming has heard the plaintive cry "my program is correct, but it still won't run". This is a fallacy. The program is incorrect. It is important to realize that a program which obeys all the rules of Basic may still contain errors.

Execution errors arise when a program attempts something illogical or impossible, even though it is syntactically valid. Typical examples are division by zero (error no. 11) and trying to use a subscript outside the bounds set for an array (error no. 9). These errors are detected when the program is run, whereupon it is halted and an error message is printed out. (If you are really unlucky your program may just 'blow up' and 'crash' the system!)

If your program crashes, requiring an ESCape to restore the system to operation, you will not get the error message you need to help pinpoint the source of trouble. But even when you do know exactly where the program stopped you may need to track down the source of the problem elsewhere. Quite possibly the statement that failed is correct, but because another part of the program did not do its job properly it cannot do what it is supposed to. A common example is the 'DATA Exhausted' condition (error 4). This may signal an error at a line containing a perfectly valid READ statement, because a DATA statement somewhere else in the program is incorrect or missing.

As a general rule, if the error is complex enough to puzzle you, the program's author, it will also confuse the computer; so you must be prepared for diagnostic messages that lead you astray if taken at face value.

9.3.3 Logical errors

Logical errors do not cause a program to fail, but make it produce the wrong answers. Generally speaking they are symptomatic of poor design. Unfortunately there is nothing to guarantee that a program consisting entirely of valid Basic statements which runs successfully to completion is actually producing the right results. So after assiduously removing all the syntax errors and making the changes necessary to let it run you may be left with a program whose output is waste paper. What do you do then?

The first thing is to take a deep breath and wrench yourself away from the computer. Invite another member of your family to play their favourite alien-zapping game on it. This will ensure that you do not succumb to the almost overwhelming urge to dive into the program and hack away like a peasant with a machette. That will only confuse the situation still further.

It is time for thinking, not hacking. So get a listing of the program, a printout of its results and a copy of the test data used and retire to a quiet corner to ponder your next move. (Test data? You never heard of that? Well, it is about time you did.) Only return to the computer when you have a firm hypothesis that could explain the program's misbehaviour and you know how your are going to test it.

9.3.4 A debugging strategy

In this book 'debugging' is a dirty word. It is something to be ashamed of. You do not boast about all-night debugging sessions to your cronies, because that it is evidence that your program was poorly designed. However, we sometimes do things we are not proud of, and if you insist on debugging at least you can do it methodically.

There are three stages in getting rid of a logical error: firstly detecting its

existence; secondly finding where it is; and thirdly putting it right. None of these need be very simple.

The first stage involves thorough testing. Devising really stringent test cases (and checking the output they lead to) is hard work. Very often it is skimped. Surprisingly many programs appear on the market from reputable suppliers simply riddled with bugs that escaped quality control. The customers are then left to do the field testing.

Since the authors of programs tend to be blind to their faults, a good rule of thumb is simply to show your work to a colleague or friend, or better still to let someone else use it for a while. You will be amazed how protective you have been towards your brainchild – never subjecting it to invalid input, never giving it too many or too few items of information, making all sorts of allowances (even without realizing it) for its shortcomings and, in brief, only testing it on the restricted class of data you already knew it could handle. A little rough treatment will do it the world of good. Schoolchildren are particularly adept at testing software 'to destruction', so if you can persuade a 12-year-old to have a bash at your program its weak spots will soon become glaringly apparent – though informal trial by ordeal is still no substitute for systematic testing of modules as they are created.

Having found something amiss you still need to know what part of the program is responsible. The art of debugging lies in forcing the program to reveal its behaviour patterns – which variables are being changed, which are not being changed, which paths are being selected, which are not being selected, and so on. If the program crunches to a halt you can use Basic's PRINT statement in direct mode to examine the values of key variables. But in general you may have to insert additional PRINT statements to display information that the program would not normally print out since, as mentioned earlier, the place where the program fails may not be the place where it went wrong. You can also use TRON and TROFF, but that is a 'scatter-gun' technique which produces mounds of indigestible output unless you know where to concentrate your search.

The tricky bit is to decide where to start looking. Here, as elsewhere, 'divide and conquer' is a good rule. If a program or module is misbehaving it must be going wrong in the first half or the second half. (It could be wrong all through, but let's not be too despressing.) So you make it display some intermediate results about half way through. "Aha!" you exclaim: "it got this far all right" or "Oh dear! It has gone wrong already". (Some programmers believe that stronger language aids the solution process, but this has yet to be scientifically proven.)

In either case, you can now repeat the search in the top or bottom half, using the same principle as the Method of Bisection (Chapter 4), and thereby halving the area of uncertainty at each stage. Finally, when you have narrowed it down to a single statement, the error – which you have probably

looked at several times before but somehow never noticed – leaps out and bites you like a cornered rat!

Incidentally, do not feel that time spent not finding errors is entirely wasted. You have made some progress when, as the police say, a suspect has been 'eliminated from enquiries'.

Finding an error is one thing: getting rid of it is quite another matter.

Broadly speaking, you will experience one of two contrasting emotions when, after diligently hunting, you finally trace a logical error to its root cause – irritation because you are amazed you overlooked something so trivial, or a sinking sensation as you realize a gross oversight in your original design. Let us consider the more problematical alternative, when there is no straightforward 'fix' because by fixing this one error other inadequacies will be exposed. What this means is that your program does not solve the problem it was intended to solve. This is difficult to admit, and we are all prone to spend to much effort delaying the admission as long as possible, even if it means an interminable series of fixes, each one generating the need for another. Yet if your program is flawed you must be prepared to go back to the drawing board and re-examine your original assumptions. There is no point in throwing good money after bad. The reasonable attitude is to salvage as much as can be salvaged (those routines that do work) and throw away the rest. It will be easier to write the second time round because your understanding of the task has increased. Some programs are just ill-fated and have to be scrapped. Make sure you know when the time has come to cut your losses and start afresh.

Finally, a word about back-up and recovery: even a perfect system cannot work after somebody has soaked the master disc in coffee. Therefore your programs must dump security copies of important information from time to time, and be able to catch up to the state they were in before the blunder occurred without re-entering months' of input. In other words you must anticipate mistakes on the part of the user. As a rough estimate, a truly robust software system will devote 80% of its work to protecting the users from themselves.

Two good sources of further reading on these subjects are *Software Design for Microcomputers* (Ogdin, Prentice–Hall, 1978) and *A Guide to Good Programming Practice* (Meek, Ellis–Horwood, 1980).

9.4 Basic with style

Programming is a craft. As you gain experience you will want to impose your own standards of craftsmanship. The recommendations we have made in the preceding three sections – on program structures, data representation and debugging – are ignored at your peril. But even within the disciplines of structured programming there is plenty of scope for individuality.

You have examples in this book of two different author's styles. Some decisions are largely a matter of taste. We both believe in descriptive variable names but we do not choose the same names. One of us prefers to use integer variables whenever whole numbers are being used, the other does not always consider it worthwhile. We prefer lower case variable names on the whole because it helps keywords to stand out visually, thus revealing the statement structure, but both of us have on occasions found reason to employ capitalized variable names. We both like REMs to stand out, but have different ways of doing it.

These are legitimate areas for personal flourishes, so we make no apology for presenting the reader with a mixture of programming styles. In fact we hope it will prove instructive. But there are certain points of style which are less a matter of personal preference: these are to do with overcoming the deficiencies of Basic as a structured programming language.

9.4.2 Structural weaknesses in Basic

Amstrad Basic is a big improvement on the original Basic of the 60s, and on some versions that are still around in the 80s, but it is by no means the last word. It does not, for instance, go as far in the right direction as Comal, a structured form of Basic defined in 1980 by Borge Christiansen in Denmark. It has two particular limitations as a structured language.

In the first place, the IF–THEN–ELSE can only be used on a single line. There is no ENDIF to close the decision structure, as there is in some more advanced languages. It would be better if one could write

```
IF      condition  THEN
        statement
        statement
        ....
ELSE
        statement
        statement
        ....
ENDIF
```

without having to cram everything into 255 characters. After all, WHILE/WEND can be used in a similar way to bracket together a series of statement lines. However, Locomotive Basic cannot handle this kind of structural grouping for conditionals. The best way round the problem is to define a couple of procedures, one for the THEN branch and one for the ELSE.

```
IF condition THEN GOSUB 1000 ELSE GOSUB 2000
```

A second structural weakness is that you cannot avoid the use of global variables as there is no way of having true parameters or LOCAL variables,

i.e. variables that are private to the routine and have no connection with variables of the same name in other modules of the program. Nevertheless you can take sensible precautions: (1) adopt a sensible naming convention (e.g. that single-letter variables are 'scratch pads' whose values cannot be relied upon to remain the same over a subroutine call); and (2) comment all uses of global variables in subroutines with REM statements.

Some of these ideas are illustrated in Listing 9.2.

Listing 9.2 Recursive pattern

```
10 REM ******************************
11 REM ** Listing 9.2 :             **
12 REM ** RECURSIVE PATTERN         **
15 REM ******************************
60 :
100 REM -- Recursive Squares:
110 INPUT "Square size ", r
120 MODE 1: BORDER 23
122 GRAPHICS PAPER 0
125 DIM x(20),y(20),size(20)
130 d%=0 : REM depth marker
140 size(1)=r
150 x(1)=INT(RND*100)+256
160 y(1)=INT(RND*100)+120
170 REM -- top-level call:
200 GOSUB 1000 : REM sq. routine
220 END
250 :
1000 REM -- Recursive square drawing routine:
1001 REM d% depth counter (for stacking)
1002 REM x(), y(), size() pseudo-parameters
1005 REM r is altered in routine.
1010 d%=d%+1: IF d%>20 THEN STOP
1020 IF size(d%)<7 THEN d%=d%-1: RETURN
1030 REM -- one more level down:
1040 MOVE x(d%),y(d%)
1050 r = size(d%)
1055 GRAPHICS PEN d%
1056 IF d% MOD 4 = 0 THEN GRAPHICS PEN 1
1060 DRAWR -r,r
1070 DRAWR r,r
1080 DRAWR r,-r
1090 DRAWR -r,-r
1100 REM -- plant parameters for lower-level calls:
1110 size(d%+1)= r/2
1120 x(d%+1)=x(d%)-r: y(d%+1)=y(d%)-r
1130 GOSUB 1000 : REM 1st recursive call.
1140 y(d%+1)=y(d%)+size(d%)
1150 GOSUB 1000
1160 x(d%+1)=x(d%)+size(d%)
1170 GOSUB 1000
1180 y(d%+1)=y(d%)-size%(d%)
1190 GOSUB 1000
1200 d%=d%-1 : REM now ascend again.
1220 RETURN
```

Recursive subroutines in a language like Amstrad Basic which does not properly support recursion by providing procedural parameters and local

variables are generally considered too hard to bother with. The trouble is that each copy of the subroutine needs its own copies of the values it is working with. Using global variables in a simple-minded way would mean that lower-level calls of the routine would scratch out information needed at higher levels.

However, once you know the technique, it is possible to do for yourself in Basic what other languages do for you 'behind the scenes'. The essential concept is the stack, as mentioned in Section 9.2.3. A stack is a data structure where all accesses – additions and removals – take place at one end, the front. It imitates a spring-loaded pile of plates in a fast-food cafeteria.

By stacking and unstacking values for the plotting routine (at line 1000) as the recursion descends and rises, we ensure that the values in **x(d%)**, **y(d%)** and **size(d%)** are appropriate to the current level of working. The level is defined by **d%** (a global variable, inevitably) which is incremented on entry to the subroutine and decremented on exit. This technique is quite general, and a useful way of helping to overcome one of Basic's greatest weaknesses.

The program draws pretty pictures by repeatedly superimposing smaller squares on top of larger ones and changing colours as the size decreases. The basic shape is drawn together with smaller shapes of the same kind. Although entirely predictable, it is unexpectedly pleasing to the eye. By altering the basic shape (here simply a square, drawn in lines 1060 to 1090) you can experiment with a whole family of different but related designs. (Try a triangle first; then think of some other shapes.)

9.4.2 Working with Basic

We finish this chapter with a short list of practical tips. Basic does impose a number of constraints, but you can learn to live with them. Good programming style, as we have tried to explain, attempts to make the most of the resources of the language and minimize its defects. The following points should help you work with Basic rather than against it; thereby making it work for you, rather than against you.

1. Avoid spaghetti programs – those so littered with GOTOs that it is impossible to perceive the logical flow. (Lasagne is far healthier!)

2. Pick line numbers with care. Start important sections of program on hundreds or thousands. If you must use GOSUB, the GOSUB 2500 is likely to mean more than GOSUB 1789. (Yes we know about the French Revolution, but you could be guillotined for lesser offences in those days.) Incidentally, this suggests that you do not go in for RENUMbering as a matter of course.

3. Choose variable names that describe their purpose. Long variable names do take up a little more space than short ones but to stick to one or two character names is to place yourself under an unnecessary disadvantage.

4. Resist the temptation to compose programs *ad lib* on the keyboard. Plan them in advance. With large programs, try out a scaled-down version before embarking on the production model.

5. Do not throw away the listing of a working program as soon as you have 'improved' it. You may need to backtrack later. This implies that a printer is the first peripheral you will want to buy.

6. Force yourself to insert REM statements and blank spaces and blank lines when typing in a program: you will be glad of it later. You can always get a cruncher program to squeeze them out later if you are really worried about speed or space-saving.

7. Eschew clever tricks that may save milliseconds of processor time or a few bytes of coding but waste many hours of human effort because they are so obscure. The best style is the simplest.

8. Ensure that your algorithms work on limiting cases such as empty files or arrays with only one element, not just on ordinary run-of-the-mill data, and that they take precautions against input they cannot deal with. They should say when they cannot handle some input, rather than just grinding to a halt.

Lastly, a word about attitude: some readers may have been surprised by how often words like 'encouragement', 'temptation', 'frustrated' and 'patience' cropped up in this chapter. This is no accident. Although programming is commonly viewed as a coldly rational process devoid of emotional content, programmers are all animals like the rest of us. By approaching the job with the right attitude you can behave more like a rational animal than an irrational one, and more like a human than a machine.

10

Case study

BASIC IN ACTION

The last chapter amounted to a sermon on the commandment "thou shalt not write ill-structured programs". Now it is time for us to practise what we preach, and apply our recommendations by working through a reasonably-sized example.

Programming case studies in textbooks are inevitably somewhat artificial, like still frames taken from a moving picture. There is never space to describe all the fits and starts of the design process. Nor would most readers have the time to plough through the intimate confessions of a software designer in unexpurgated detail.

Nevertheless we believe it is important to take a non-trivial program from concept to completion, giving a blow by blow account of its progress, even including a few backward steps. That way you can, as it were, look over the programmer's shoulder and gain some insight into the craft of programming.

10.1 The plan

Our objective is to write a program that plays a code breaking game based on 'Mastermind' (copyright 1972, Invicta Plastics Ltd.). It is a small enough problem to fit in the space, and time, available; but it is not trivial, so it will call upon some of the disciplines of structured programming outlined in Chapter 9. It has certain other points in its favour too.

It provides the opportunity for a splash colour.
It does not need discs, so our tape-based readers can profit from it.
It requires careful thought about solution strategies.
It provides a framework around which many readers will be able to develop other games of their own.

Above all, it is fun to play.

To recap, briefly, on the rules. The game is played by two players. One makes a secret 'code' which is a combination of 4 of the 7 permitted colours,

and the other player tries to guess this code. The seven colours to pick from are: Red, Green, Blue, Magenta, Cyan, Pink and Yellow. (On screen Cyan appears as a lighter blue than Blue.) A code is represented by a row of four coloured pegs. Any four colours may be used from the seven, and repeats are allowed, so

Red Red Red Red
Blue Green Blue Cyan
Red Yellow Green Blue
Magenta Magenta Pink Pink (Yuk!)

are all permissible codes.

The guessing player makes a guess, a four-colour combination, which the code-maker scores in terms of 'Blacks' and 'Whites'. A black is given for a peg of the correct colour in the correct place; a white is given for the correct colour in the wrong place. Note that no peg may be scored more than once, so the number of blacks plus whites should equal the number of correct colours, ignoring position. Four blacks means a correct guess.

Ten guesses is the maximum allowed. Players take turns in being code maker and code breaker.

We shall require the program to play a dual role, not just making codes for the user to break and scoring the user's guesses, but also breaking codes itself. Code breaking is the more demanding part, since for people it requires some reasonably challenging brainwork.

But there is no need for us to delve deeply into Artificial Intelligence, as long as we recognize one point, which will be the key to the machine's code-breaking strategy: the colour-codes can be interpreted numerically.

A row of four pegs from seven colours can be regarded as a 4-digit number expressed to the base of 7. We use decimal numbering in daily life, but computer people are also accustomed to counting in the hexadecimal (base 16), octal (base 8) and binary (base 2) number systems.

Base 7 is not such a peculiar idea. Instead of the columns denoting units, tens, hundreds (10 × 10), thousands (10 × 10 × 10) and so on, they denote units, sevens, 49s (7 × 7), 343s (7 × 7 × 7) and so on. By coding our seven colours numerically

0 red
1 green
2 blue
3 magenta
4 cyan
5 pink
6 yellow

we can interpret any code such as

blue blue green cyan

as the expression of a base-7 number. In this case the number is 2214, which has the decimal value of 795 = 2 × 343 + 2 × 49 + 1 × 7 + 4 = 686 + 98 + 7 + 4. (We could have chosen other colouring schemes; but this one gives a fair range of hues which are not easily confused – unless you have a monochrome screen.)

Where does this excursion into base-7 arithmetic get us?

Well, one thing it tells us is that there are only 2401 possible codes, since 343×6 + 49×6 + 7×6 + 6 = 2400. The computer can maintain a list of all the 2401 possibilities and simply cross off the ones that do not fit the facts as each guess is scored. A human would not want to keep track of 2401 different combinations, but for the machine this is quite straightforward.

At every stage in the game the computer needs to know whether each combination is still possible or not. This is just one bit of information: it could be represented as 0 or 1, so in theory all we need to do is set aside 2401 bits or just over 300 bytes of memory to keep track of the status of the game. In practice it is far simpler to avoid bit-twiddling and use a whole integer (16 bits) for each possibility. Then by declaring

DIM code%(2400)

we can set up an array where each location records the status of a particular four-colour combination. Thus, for instance, the status of the combination

cyan red yellow blue

will be held at location

4 × 343 + 0 × 49 + 6 × 7 + 2 = 1416

in the vector. We will use zero to indicate that a code is not yet ruled out by the information so far and one to indicate that it is no longer possible.

Thus our solution strategy is a method of elimination.

What we have done is make a decision about data structuring before even starting on the program design. This is quite normal, and indeed desirable. Before going further, however, it is as well to point out that there are other ways of doing it. A fixed 2401-item vector has certain disadvantages. For example, what if we wanted eight colours instead of seven? Then we would need 4096 locations, and it might be worth reducing this to 512 bytes by bit-packing. With ten colours we would need 10 000 locations. At some point the system would cease to be practicable, taking too much memory space and too much time to process. But then again, the game itself would be become unwieldy; and there are really only about 8 distinguishable colours, discounting 'flashing colours', to display on the Amstrad microcomputer anyway.

For the moment we will press ahead with this key decision. If it works, we can stick with it; if (heaven forbid!) it proves impractical, we must be prepared to return and alter it. That means that we ought to hide the details of the data representation as far as possible in case we have to change it later.

We will also make a scheduling decision (see Section 9.1.4). We will split up the implementation into two main phases: firstly we will do the simpler task, getting the computer to set the code and the user to solve it; only when that is done will we implement the harder part, making the computer solve our codes. The third phase – enhancement of the game's performance – will be left as an exercise for the reader!

The first phase allows us to create a framework for the program and test a number of essential subroutines before tackling the difficulties of automating the solution process. In effect, we postpone the tricky bits until we are good and ready for them.

Finally, we make one practical decision, to use Mode 0 for the screen display rather go for high-resolution graphics. Actually this is forced on us by the need for at least seven distinct colours. It also means that the character shapes used to represent the pegs are nice chunky patches of colour.

10.2 The program: version 1

Now we can proceed to design the program. We start at the top, with the main line.

```
200  MODE 0
220  DIM code%(2400)
230  GOSUB 2300 : REM initialization
300  PRINT "Do you want to start first (Y/N)? ";
310  GOSUB 2000 : h% = yeah%
315  :
320  htot%=0 : ctot% = 0 : REM total guesses.
322  c% = 0 : g% = 0 : REM games played.
323  endgame% = 0
325  REM -- main loop:
330  WHILE endgame% = 0
333      REM -- person making code:
340      IF h% THEN GOSUB 2200: GOSUB 4000: g%=g%+1
360      REM -- computer making code:
370      CLS
380      GOSUB 2500 : REM own turn
390      h% = 1
395      REM always TRUE after 1st cycle.
400      c% = c% + 1
410      GOSUB 3700 : REM score-keeping routine
```

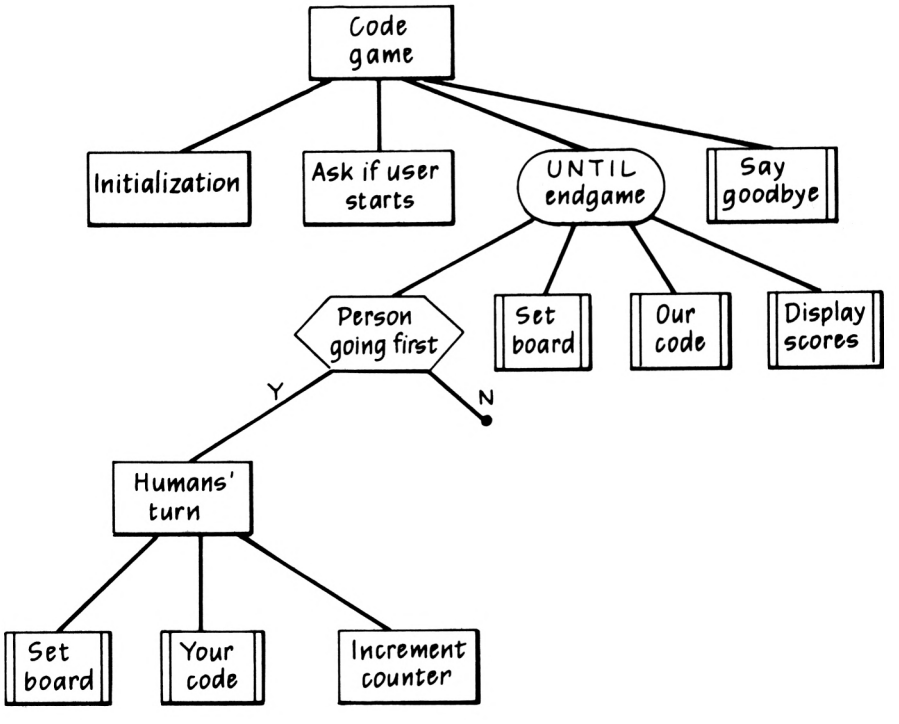
```
420       PRINT "Stop now (Y/N) ? ";
430       GOSUB 2000 : endgame% = yeah%
440       WEND
450  PRINT
460  GOSUB 5500 : REM Say goodbye
470  END
480  :
```

This is complete top-level outline. The guts of the system is lines 330 to 440. This is a loop in which each side takes turns to set the code. The two critical routines are

```
4000  -- computer guesses person's code
2500  -- person guesses computer's code
```

and we will ignore the first one completely until the second is fully operational. We can still test the logic of the program because on the first cycle round the loop the person can opt to skip the code-making part. The logic is also presented diagrammatically in Fig. 10.1.

Fig. 10.1 Diagram of code-game logic



The other undefined routines are as follows.

2300 global initialization
2200 prepares for a new game
3700 displays the latest score for both sides.
2000 accepts a Yes or No answer only
5500 signs off.

None of these presents serious complications. The interesting part is subroutine 2500. This will have to select a combination for the code at random (fairly simple) and then go into a loop, obtaining the user's next guess and scoring it until the user gets it right or has had too many goes. We have set the limit at ten attempts.

Obtaining the user's guess is not conceptually complex, but it requires some thought about the method of input and output. Users do not, on the whole, take kindly to having to type words like

magenta

without spelling mistakes. So we will let the user compose the latest guess on screen, one peg at a time, using single keystrokes. This only requires use of three keys.

SPACE BAR	change to the next available colour.
RETURN	move on to the next peg (or finish after the 4th).
DELETE	go back to the previous peg.

and has the advantage of visual feedback. The DELETE option allows the user to change his mind or correct mistakes. The SPACE bar lets the user cycle through the colours one by one. Therefore it does not require memorization of what hues are available. As we pointed out in Chapter 6, it is a good idea only to present the user with valid choices when this can be arranged, and here it can. The RETURN key moves through the pegs from left to right. After the 4th peg, pressing RETURN causes acceptance of the current guess.

Scoring the user's guess in terms of blacks and whites will require two additional routines, one to count exact matches (right colour in the right place) and one to count partial matches (right colour in any place). The scoring functions must take care not to count one peg twice. For example, if there is one red in the code and two in the guess (one in the correct position) the user only gets a black, not a white as well.

Here then is the listing of the program at the end of the first day.

Listing 10.1 Codegame, Version 1

```
10 REM ******************************
11 REM ** Listing 10.1 :           **
12 REM ** CODE GAME, VERSION 1     **
15 REM ******************************
60 :
100 REM -- Code-breaking Game:
110 MODE 0: BORDER 15
200 size%=2400 : REM main array limit
210 DIM hues$(6) : REM colour names
220 DIM code%(size%), a%(4),x%(4),y%(4)
230 GOSUB 2300 : REM initialization
240 PRINT "Welcome to the code-making game:"
250 PRINT
260 PRINT "Would you like some instructions (Y/N)? ";
270 GOSUB 2000: h%=yeah%
280 IF h% THEN CHAIN "INSTRUCT"
290 PRINT
300 PRINT "Do you want to start first (Y/N) ? ";
310 GOSUB 2000 : h%=yeah%
320 :
330 htot%=0: ctot%=0: c%=0: g%=0
340 endgame%=0
350 REM -- Main loop:
360 WHILE endgame%=0
370   REM -- Person making code:
380   IF h% THEN GOSUB 2200: GOSUB 4000 : g%=g%+1
390   REM -- Computer making code:
400   CLS
410   GOSUB 2500 : REM own turn
420   h%=1 : REM always TRUE from now on.
430   c%=c%+1
440   GOSUB 3700 : REM scoring
450   PRINT "Had enough (Y/N) ? ";
460   GOSUB 2000 : endgame%=yeah%
480   WEND
500 PRINT
520 GOSUB 5500 : REM goodbye
550 END
999 :
1000 REM -- Colour data:
1010 DATA red,green,blue,magenta,cyan,pink,yellow
1100 :
2000 REM -- Yes/No Routine:
2010 yeah%=88 : REM neither true nor false
2020 WHILE yeah%<>0 AND yeah%<>1
2030   y$=INKEY$: IF y$="" THEN GOTO 2030
2040   IF y$="Y" OR y$="y" THEN yeah%=1
2050   IF y$="N" OR y$="n" THEN yeah%=0
2060   WEND
2070 PRINT y$
2080 RETURN
2090 REM yeah% contains result (1 or 0).
2100 :
2200 REM -- Set-board routine:
2220 CLS
2230 FOR i%=0 TO size%
2240   code%(i%)=0 : NEXT
2250 RETURN
2280 :
2300 REM -- Initialization routine:
2310 blob$=CHR$(143) : REM peg character
2320 base%=1
2330 FOR i%=0 TO 6
2340   READ hues$(i%) : REM hue-names
2360   NEXT i%
2370 INK 0,1 : PAPER 0
2380 INK 1,6
2390 INK 2,9
2400 INK 3,2
2410 INK 4,8
2420 INK 5,23
```

```
2430 INK 6,16
2440 INK 7,24
2450 REM inks set to correspond with names.
2460 INK 8,22
2470 RETURN
2480 :
2500 REM -- Own-turn routine:
2510 PRINT: PRINT "Computer making code :"
2520 PRINT
2530 goes%=0
2540 FOR p%=1 TO 4
2550   a%(p%) = INT(RND*7) : REM 0 to 6
2560   NEXT p%
2570 b%=0 : REM no. of 'blacks'.
2580 WHILE b%<4 AND goes%<10
2590   goes%=goes%+1
2600   GOSUB 2800 : REM get guess
2610   REM guess now in y%().
2615   GOSUB 3500 : REM copy into x%()
2620   GOSUB 3300 : REM show code
2630   GOSUB 3000 : REM exact matches
2640   GOSUB 3100 : REM partial matches
2650   b%=matches% : w%=partial%-matches%
2660   GOSUB 3600 : REM feedback
2670   WEND
2680 GOSUB 3200 : REM message
2690 htot%=htot%+goes%
2700 RETURN
2720 :
2800 REM -- Get-guess routine:
2810 REM puts human guess into y%():
2820 vert%=goes%*2 + 2
2830 FOR pegs%=1 TO 4
2840   guessed%=0: x%=99
2850   WHILE x%<>13
2855     PEN 8
2860     LOCATE 1,vert%: PRINT goes%;
2870     horz%=pegs%*2 + 4
2880     LOCATE horz%,vert%
2888     x%=99
2890     PEN guessed%+1: PRINT blob$
2900     WHILE x%<>13 AND x%<>32 AND x%<>127
2910       c$=INKEY$: IF c$="" THEN GOTO 2910
2920       x%=ASC(c$)
2925       WEND
2930     IF x%=32 THEN guessed%=guessed%+1
2935     CLEAR INPUT
2940     REM space steps onto next hue.
2950     IF x%=127 THEN pegs%=pegs%-1: PRINT CHR$(7); : REM delete
2960     IF guessed%>6 THEN guessed%=0
2970     WEND
2980   y%(pegs%)=guessed%
2990   NEXT pegs%
2995 PEN 8 : RETURN
2999 :
3000 REM -- Exact-matching routine:
3010 REM a%() is answer, y%() is guess.
3020 matches%=0
3030 FOR em%=1 TO 4
3040   IF a%(em%)=y%(em%) THEN matches%=matches%+1
3050   NEXT
3060 RETURN
3070 REM with matches%.
3080 :
3100 REM -- Partial-match routine:
3110 partial%=0
3120 FOR pm%=1 TO 4: jj%=1
3130   WHILE jj%<=4
3140     IF a%(pm%)=y%(jj%) THEN partial%=partial%+1: y%(jj%)=99: jj%=4
3150     jj%=jj%+1
3160     WEND
3170   NEXT
3180 RETURN : REM with partial%
3190 :
3200 REM -- Message routine:
```

```
3210 REM end of human´ guessing:
3220 IF b%<4 THEN PRINT "You failed !!";CHR$(7) ELSE PRINT "Got it in";goes%;"go
es."
3230 PRINT: PRINT "The answer was:"
3240 FOR ii%=1 TO 4
3250   p%=a%(ii%): PEN p%+1
3260   PRINT blob$;" ";hues$(p%)
3270   NEXT
3280 PEN 8
3290 RETURN
3295 :
3300 REM -- Show-code routine:
3310 vert% = goes%*2 + 2
3320 FOR ip%=1 TO 4
3330   horz%=ip%*2 + 4
3340   jj%=x%(ip%): PEN jj%+base%
3350   LOCATE horz%,vert%
3360   PRINT blob$;" ";
3370   NEXT
3380 PRINT : PEN 8
3390 RETURN
3400 :
3500 REM -- Save-copy routine:
3510 REM makes copy of y%() in x%():
3520 FOR i%=1 TO 4
3530   x%(i%)=y%(i%)
3540   NEXT
3550 RETURN
3560 :
3600 REM -- Feedback routine:
3610 REM vert% is global:
3620 LOCATE 1,vert%+1
3630 PEN 15 : PRINT "Blacks:";b%;
3640 PEN 8 : PRINT "Whites:";w%
3650 RETURN
3660 :
3700 REM -- Scoring routine:
3710 REM shows average score of both players:
3720 REM globals: c%,g%,ctot%,htot%
3730 IF c%<1 OR g%<1 THEN RETURN
3740 REM quit if human hasn´t tried.
3750 them = htot / c%
3760 us = ctot% / g%
3770 PRINT
3780 PRINT TAB(8);"Games";TAB(16);"Score"
3790 PRINT "You";TAB(8);c%;TAB(16);them
3800 PRINT "Me";TAB(8);g%;TAB(16);us
3810 PRINT CHR$(7)
3820 IF us < them THEN PRINT "I´m winning!!"
3830 IF us > them THEN PRINT "You´re ahead!"
3840 PRINT
3850 RETURN
3880 :
```

At this point it could set a code for the user to guess and score the user's guesses correctly.

The most important procedure is on lines 2500 to 2700. This uses a subroutine, at lines 2800 to 2995, which obtains the user's next guess in the manner described above. It also uses two matching routines

```
3000    : REM exact matches
3100    : REM partial matches
```

to compare the actual code with a guess.

The exact match routine is simple: it just steps through the two arrays **a%()** and **y%()**, incrementing a counter where they are equal. The partial matching function is a little more involved. It contains a WHILE/WEND loop

within a FOR loop, to go through the second array for each position in the first one. The inner loop stops when it finds a match or when it has looked at all four items in **y%**. If it does find a match, it actually changes the contents of the matching location (line 3140) to make sure that no peg is counted more than once. So the **y%()** array may be altered as a result, which is why it is copied into **x%()** for later use (line 2615).

The various ink colours are set up on lines 2370 forwards. An eighth colour (pastel green) is selected for system messages, and a ninth (orange) for the border. If you look at the initialization routine, you will see that line 2310

```
blob$ = CHR$(143)
```

sets up the image of a peg. This is one of the Amstrad's block-graphic characters – just a rectangular block. The words naming each colour are stored in the string array **hues$()**.

At this stage the **code%()** array is not in use, but three small arrays **a%**, **x%**, and **y%** are declared on line 220 for holding the answer and the guess and for temporary working storage.

10.3 The finished program [CODEGAME]

Now we come to the hard part – making the computer guess codes for itself.

The reasoning behind the method adopted is as follows. When a guess is scored, say with 1 black and 1 white, it is compared for exact and partial matches with the unseen code. Therefore by comparing a guess with any possible combination you can determine whether that combination could conceivably be the answer. Assume that combination is the answer, and score the guess accordingly: if it gets a different score from the one actually awarded it could not possibly be the answer; if it receives the same score, it still could be. Each time a guess is made and scored a large number of candidates for the hidden code can be eliminated, rapidly converging on the correct answer (so long as the user computes the score accurately).

A worked example may make this plainer.

CODE :	Blue	Blue	Cyan	Magenta
	1	2	3	4
GUESS :	Blue	Green	Blue	Red

SCORE : Blacks = 1; Whites = 1
[Blue 1 : Blue 1 ; Blue 2 : Blue 3]

POSSIBILITY 1:	Blue	Green	Green	Green
	1	2	3	4
GUESS :	Blue	Green	Blue	Red

SCORE : Blacks = 2 ; Whites = 0
[Blue 1 : Blue 1; Green 2 : Green 2]

POSSIBILITY 2:	Blue	Yellow	Yellow	Green
	1	2	3	4
GUESS :	Blue	Green	Blue	Red

SCORE : Blacks = 1 ; Whites = 1
[Blue 1 : Blue 1 ; Green 4 : Green 2]

The first possibility cannot be the secret code because if it were the score for that guess would have been different, 2 and 0 instead of 1 and 1. So it can be eliminated from further consideration. The second possibility cannot be ruled out yet – even though it is in fact not the answer.

Let us now look at the program at the end of the second day to see how these ideas are put into practice.

Listing 10.2 Codegame, Version 2

```
10 REM ******************************
11 REM ** Listing 10.2 :           **
12 REM ** CODE GAME, VERSION 2     **
15 REM ******************************
60 :
100 REM -- Code-breaking Game:
110 MODE 0: BORDER 15
200 size%=2400 : REM main array limit
210 DIM hues$(6) : REM colour names
220 DIM code%(size%), a%(4),x%(4),y%(4)
230 GOSUB 2300 : REM initialization
240 PRINT "Welcome to the code-making game:"
250 PRINT
260 PRINT "Would you like some instructions (Y/N)? ";
270 GOSUB 2000: h%=yeah%
280 IF h% THEN CHAIN "INSTRUCT"
290 PRINT
300 PRINT "Do you want to start first (Y/N) ? ";
310 GOSUB 2000 : h%=yeah%
320 :
330 htot%=0: ctot%=0: c%=0: g%=0
340 endgame%=0
350 REM -- Main loop:
360 WHILE endgame%=0
370   REM -- Person making code:
380   IF h% THEN GOSUB 2200: GOSUB 4000 : g%=g%+1
390   REM -- Computer making code:
400   CLS
```

```
410   GOSUB 2500 : REM own turn
420   h%=1 : REM always TRUE from now on.
430   c%=c%+1
440   GOSUB 3700 : REM scoring
450   PRINT "Had enough (Y/N)? ";
460   GOSUB 2000 : endgame%=yeah%
480   WEND
500 PRINT
520 GOSUB 5500 : REM goodbye
550 END
999 :
1000 REM -- Colour data:
1010 DATA red,green,blue,magenta,cyan,pink,yellow
1100 :
1500 REM -- Splitting routine:
1510 FOR z%=4 TO 1 STEP -1
1520   y%(z%)=nx% MOD 7: nx%=nx%\7 : NEXT z%
1530 RETURN
1540 REM splits nx% into 4 base-7 digits in y%().
1550 :
2000 REM -- Yes/No Routine:
2010 yeah%=88 : REM neither true nor false
2020 WHILE yeah%<>0 AND yeah%<>1
2030   y$=INKEY$: IF y$="" THEN GOTO 2030
2040   IF y$="Y" OR y$="y" THEN yeah%=1
2050   IF y$="N" OR y$="n" THEN yeah%=0
2060   WEND
2070 PRINT y$
2080 RETURN
2090 REM yeah% contains result ( 1 or 0).
2100 :
2200 REM -- Set-board routine:
2220 CLS
2230 FOR i%=0 TO size%
2240   code%(i%)=0 : NEXT
2250 RETURN
2280 :
2300 REM -- Initialization routine:
2310 blob$=CHR$(143) : REM peg character
2320 RESTORE : base%=1
2322 RANDOMIZE TIME
2330 FOR i%=0 TO 6
2340   READ hues$(i%) : REM hue-names
2360   NEXT i%
2370 INK 0,1 : PAPER 0
2380 INK 1,6
2390 INK 2,9
2400 INK 3,2
2410 INK 4,8
2420 INK 5,23
2430 INK 6,16
2440 INK 7,24
2450 REM inks set to correspond with names.
2460 INK 8,22
2470 RETURN
2480 :
2500 REM -- Own-turn routine:
2510 PRINT: PRINT "Computer making code :"
2520 PRINT
2530 goes%=0
2540 FOR p%=1 TO 4
2550   a%(p%) = INT(RND*7) : REM 0 to 6
2560   NEXT p%
```

```
2570 b%=0 : REM no. of 'blacks'.
2580 WHILE b%<4 AND goes%<10
2590   goes%=goes%+1
2600   GOSUB 2800 : REM get guess
2610   REM guess now in y%().
2615   GOSUB 3500 : REM copy into x%()
2620   GOSUB 3300 : REM show code
2630   GOSUB 3000 : REM exact matches
2640   GOSUB 3100 : REM partial matches
2650   b%=matches% : w%=partial%-matches%
2660   GOSUB 3600 : REM feedback
2670   WEND
2680 GOSUB 3200 : REM message
2690 htot%=htot%+goes%
2700 RETURN
2720 :
2800 REM -- Get-guess routine:
2810 REM puts human guess into y%():
2820 vert%=goes%*2 + 2
2830 FOR pegs%=1 TO 4
2840   guessed%=0: x%=99
2850   WHILE x%<>13
2855     PEN 8
2860     LOCATE 1,vert%: PRINT goes%;
2870     horz%=pegs%*2 + 4
2880     LOCATE horz%,vert%
2888     x%=99
2890     PEN guessed%+base%: PRINT blob$
2900     WHILE x%<>13 AND x%<>32 AND x%<>127
2910       c$=INKEY$: IF c$="" THEN GOTO 2910
2920       x%=ASC(c$)
2925       WEND
2930     IF x%=32 THEN guessed%=guessed%+1
2933     CLEAR INPUT
2940     REM space steps onto next hue.
2950     IF x%=127 THEN pegs%=pegs%-1: PRINT CHR$(7); : REM dele
te
2960     IF guessed%>6 THEN guessed%=0
2970     WEND
2980   y%(pegs%)=guessed%
2990   NEXT pegs%
2995 PEN 8 : RETURN
2999 :
3000 REM -- Exact-matching routine:
3010 REM a%() is answer, y%() is guess.
3020 matches%=0
3030 FOR em%=1 TO 4
3040   IF a%(em%)=y%(em%) THEN matches%=matches%+1
3050   NEXT
3060 RETURN
3070 REM with matches%.
3080 :
3100 REM -- Partial-match routine:
3110 partial%=0
3120 FOR pm%=1 TO 4: jj%=1
3130   WHILE jj%<=4
3140     IF a%(pm%)=y%(jj%) THEN partial%=partial%+1: y%(jj%)=99
: jj%=4
3150     jj%=jj%+1
3160     WEND
3170   NEXT
3180 RETURN : REM with partial%
```

```
3190 :
3200 REM -- Message routine:
3210 REM end of human´ guessing:
3220 IF b%<4 THEN PRINT "You failed !!";CHR$(7) ELSE PRINT "Got
it in";goes%;"goes."
3230 PRINT: PRINT "The answer was:"
3240 FOR ii%=1 TO 4
3250   p%=a%(ii%): PEN p%+1
3260   PRINT blob$;" ";hues$(p%)
3270   NEXT
3280 PEN 8
3290 RETURN
3295 :
3300 REM -- Show-code routine:
3303 REM shows code in x%():
3310 vert% = goes%*2 + 2
3315 LOCATE 1,vert%: PRINT goes%;
3320 FOR ip%=1 TO 4
3330   horz%=ip%*2 + 4
3340   jj%=x%(ip%): PEN jj%+base%
3350   LOCATE horz%,vert%
3360   PRINT blob$;" ";
3370   NEXT
3380 PRINT : PEN 8
3390 RETURN
3400 :
3500 REM -- Save-copy routine:
3510 REM makes copy of y%() in x%():
3520 FOR i%=1 TO 4
3530   x%(i%)=y%(i%)
3540   NEXT
3550 RETURN
3560 :
3600 REM -- Feedback routine:
3610 REM vert% is global:
3620 LOCATE 1,vert%+1
3630 PEN 15 : PRINT "Blacks:";b%;
3640 PEN 8 : PRINT "Whites:";w%
3650 RETURN
3660 :
3700 REM -- Scoring routine:
3710 REM shows average score of both players:
3720 REM globals: c%,g%,ctot%,htot%
3730 IF c%<1 OR g%<1 THEN RETURN
3740 REM quit if human hasn´t tried.
3750 them = htot% / c%
3760 us = ctot% / g%
3770 PRINT
3780 PRINT TAB(7);"Games";TAB(15);"Score"
3790 PRINT "You";TAB(7);c%;TAB(15);them
3800 PRINT "Me";TAB(7);g%;TAB(15);us
3810 PRINT CHR$(7)
3820 IF us < them THEN PRINT "I´m winning!!"
3830 IF us > them THEN PRINT "You´re ahead!"
3840 PRINT
3850 RETURN
3880 :
4000 REM -- Code-breaking routine:
4010 LOCATE 1,1: PEN 8
4020 PRINT "Computer guessing :"
4025 gmax%=1666
4030 goes%=0: yeah%=0
```

```
4040 WHILE yeah%=0
4050   PRINT "Have you made a code (Y/N)? ";
4060   GOSUB 2000 : REM yes/no answer
4070   WEND
4075 CLS
4080 failure%=0 : b%=0
4090 WHILE b%<4 AND goes%<10 AND failure%=0
4100   goes%=goes%+1
4110   LOCATE 1,1: PRINT "My guess no.";goes%;": "
4120   GOSUB 4500 : REM find a code
4130   REM y%() now has computer's guess.
4140   IF failure%=1 THEN GOSUB 4400 ELSE GOSUB 4200
4145   gmax%=size%
4150   WEND
4160 LOCATE 1,22
4170 GOSUB 5200 : REM results
4180 ctot%=ctot%+goes% : REM running total
4190 RETURN
4195 :
4200 REM -- Good-turn routine:
4210 REM -- normal post-guess feedback:
4215 GOSUB 3500 : REM copy code
4220 GOSUB 3300 : REM show code
4230 w%=-1: b%=-1
4240 WHILE w%<0 OR b%<0 OR w%>4 OR b%>4 OR (w%+b%)>4
4250   LOCATE 1,vert%+1: PEN 15
4260   INPUT "Blacks"; b%
4270   PEN 8 : LOCATE 11,vert%+1
4280   INPUT "Whites"; w%
4290   WEND
4300 IF b%<4 AND goes%<10 THEN GOSUB 4800 : REM test all
4310 RETURN
4330 :
4400 REM -- Bad-turn routine:
4410 REM abnormal ending.
4420 LOCATE 1,vert%+1
4430 PRINT "No more moves left!"
4440 PRINT CHR$(7)
4450 PRINT "You blundered!"
4460 goes%=goes%-1 : REM score bonus
4470 RETURN
4480 :
4500 REM -- Find-code routine:
4510 REM puts a possible code into a%() & y%():
4520 failure%=0
4530 n%=INT(RND*size%)
4540 cc%=0 : this%=-8
4550 WHILE this%<0 AND cc%<size%+1
4560   IF code%(n%)=0 THEN this%=n%
4570   cc%=cc%+1
4580   n% = (n%+1) MOD (size%+1)
4590   WEND
4600 IF this%<0 THEN failure%=1: RETURN
4610 REM failure% is global signal.
4620 REM -- Turn guess into colour codes:
4625 nx%=this%
4630 GOSUB 1500 : REM split-up
4640 FOR n%=1 TO 4: a%(n%)=y%(n%): NEXT
4645 REM copies y%() into a%().
4650 RETURN
4660 :
4800 REM -- Routine to test possibilities:
4810 LOCATE 16,2: PRINT "    ";
```

```
4820 FOR nn%=0 TO gmax%
4830   IF code%(nn%)=0 THEN GOSUB 5000
4840   LOCATE 15,2: PRINT nn%;
4850   NEXT nn%
4860 RETURN
4880 REM counter shows that something is going on.
4890 :
5000 REM -- Move-test routine:
5005 nx%=nn%
5020 GOSUB 1500 : REM split-up into y%()
5030 GOSUB 3000 : REM matches
5040 mm%=matches%
5050 IF mm%<>b% THEN code%(nn%)=1: RETURN
5060 GOSUB 3100 : REM partial matches
5070 pp%=partial%-matches%
5080 IF w%<>pp% THEN code%(nn%)=1
5090 RETURN
5100 REM -- only combinations with the same score are still poss
ible.
5110 REM w% and b% are global.
5150 :
5200 REM -- Results routine:
5210 PRINT "I took";goes%;"guesses"
5220 IF b%<4 THEN PRINT "and still missed it!";CHR$(7)
5230 IF b%=4 THEN PRINT "to break the code."
5240 IF goes%<5 THEN PRINT "Not bad for a machine?"
5250 IF goes%>7 THEN PRINT "Well, I'm only human!!"
5260 PRINT "Hit RETURN to go on:"
5270 c$=""
5280 WHILE c$<>CHR$(13)
5300   c$=INKEY$: IF c$="" THEN GOTO 5300
5310   WEND
5320 RETURN
5330 :
5500 REM -- Farewell routine:
5510 PRINT "Thanks for playing."
5520 PRINT "Bye for now!"
5530 RETURN
5550 :
```

At line 4000 is the subroutine which plays the game with the computer as code-breaker. What it does is to call subroutine 4500 which chooses at random one of the remaining possibilities and places it as a 4-digit code in **y%**. If there are no possibilities left, it signals **failure%** and subroutine 4440 is entered. But normally subroutine 4200 is then called to display the computer's guess, get the score and update the array of possible combinations.

The latter process is carried out by a subroutine at lines 4800 to 4880, which goes through the 2401 combinations and calls the testing routine (line 5000) on all of them that are still possible.

The testing routine compares guess **a%** with possibility **nn%**. **nn%** ranges from 0 to 2400). First **nn%** is converted into four separate base-7 digits in the **y%()** array. Then the exact-match routine, which was developed in the previous phase, is applied to see whether the guess would get the same number of blacks **(b%)** as it actually did if **a%** was the answer. If not,

possibility **nn%** is eliminated and the procedure terminates (line 5050): there is no need to look at the partial matches. If the exact matches are the same, the procedure continues by invoking the partial-matching procedure. If the partial matches do not tally with **w%** (line 5080) the combination **nn%** can be crossed out. Otherwise it remains in play for the next round. In either case the routine is finished (line 5090).

This then is the 'finished' program, although it can be developed further as we shall see in the next section. It plays a better game, most of the time, than its author – taking around five guesses per game. After its first move it takes nearly four minutes (typically 220 seconds) to assimilate the score. After the second move it takes about two and a half minutes (depending on the code). Thereafter it chugs along at around 80 seconds a move. Whether you find that speed acceptable is up to you; but it is quicker than many people.

There is no printout to reproduce in the book: the program is screen-based. You will have to type it in and try it yourselves to see what it does. But here at least is a transcript of one game it played as code breaker, where it did not do spectacularly well.

Person making code :

Code :	Yellow	Yellow	Cyan	Blue		
Guess :					Blacks,	Whites
1.	Pink	Magenta	Cyan	Magenta	1	0
2.	Cyan	Red	Cyan	Red	1	0
3.	Cyan	Yellow	Green	Green	1	1
4.	Pink	Red	Yellow	Green	0	1
5.	Blue	Yellow	Cyan	Blue	3	0
6.	Blue	Yellow	Cyan	Yellow	2	2
7.	Yellow	Yellow	Cyan	Blue	4	0

The exact sequence you get depends, of course, on the random-number generator, so it may guess differently when you give it the same code to break. (But if it does not get the answer at all, you have made a typing error.)

10.4 Further developments

The program now works properly, but that does not mean it is finished. We conclude with a few critical remarks that may inspire some readers to produce better versions of their own.

Let us begin with the matter of speed, already alluded to above. Taking up to four minutes after the first guess to ponder its next move may seem a bit slow. And indeed, this is after the inclusion of line 4025 (**gmax%=1666**) whose effect is to prevent it considering all possibilities first time round, and

thereby play faster at the expense of taking more tries, on average, to find the solution. You can make the program play better by altering this to **gmax%=size%**; but then it will take six minutes to reply after its first move.

This is the point at which the dedicated hacker dives straight into machine code. But it is normally more profitable to improve the algorithm before rewriting it in another language: the improvement per unit of effort is usually greater, and the re-write often turns out to be unnecessary.

The heart of the matter is subroutine 5000. This is executed up to 2401 times, and it calls the matching routines at 3000 and 3100, which contain loops themselves. Subroutine 3000 is tried first, because it is faster, and if the current guess fails on this test, it can be eliminated without entering the slower routine at line 3100. This is sensible, but how could the move-tester be speeded up further? A small improvement would be simply to remove the REMs, although this spoils the style. It might also be worth combining lines 5020 and 5030 onto one line and saving a line number. You could also remove line 4840, but then the user would have no indication that the machine was doing anything while it was 'thinking'.

A bigger improvement can be had by re-thinking the matching routines. They are fine when being used only once each per guess, but they could still do with improvement when being repeated thousands of times. One problem is that the partial match routines re-does some work already done by the exact match routine. It would be possible for the first one to tell the second, in effect, which pegs to avoid. Another possibility would be for the partial routine to stop early if the total of matches (whites plus blacks) reached four, since there could never be more. It would also be possible – at the price of readability – to lose a few spaces and line numbers in subroutine 3100, saving some time that way. If you are prepared to put in a little thought along these lines, you should be able to halve the time taken by the elimination process without descending to machine code.

If you do get down to Assembler level, the matching routines are the ones to rewrite first.

A second point is that the program is still incomplete: it needs instructions. Line 280 CHAINs another program called INSTRUCT but this has not been written. All it needs to do is present the rules of the game and explain the method of input then CHAIN back to the main program. You could write this by referring back to Section 10.1.

A third criticism concerns the use of global variables and the choice of their names. The program has several one-letter names for global data items.

a% 4-digit answer array.
x% 4-digit workspace array.
y% 4-digit guess array.
h% Human's choice of whether to move first.

c% Games completed by computer.
g% Games completed by human.
b% No. of Blacks for latest guess.
w% No. of Whites for latest guess.

Short variable-names are marginally faster in an interpreted language like Basic; but that is rather a feeble excuse. It just shows how standards get lowered when your book is already two weeks late and Christmas is hurtling towards you like a runaway express. There are also a few constants, such as 32 and 127 in the program, for the SPACE and DELETE characters respectively, which really ought to be given names. (E.g. **spacebar% = 32**.)

Fourthly, and most important, the computer does not play a perfect game. It is more than a match for most humans (with **gmax%** set to equal **size%**), as long as you are prepared to wait; but for those readers who like a challenge, it is worth thinking about ways of optimizing its code-breaking strategy.

At present it chooses its next guess at random from those codes that are still possible. One suggestion would be for it to look ahead, although of course this would slow it down further. That is: after the first guess, it could consider all the prospective guesses from the point of view of how much information each one was likely to yield. This could be done as follows.

For each potential guess, compute what scores it could receive on the basis of the past record.
Work out for each of those possible scores how much information would be gained, i.e. how many of the remaining possibilities would be eliminated.
Average these over the possible scores for that guess and retain the average.
Select the guess with the highest average.

Here we are trying to choose guesses that have most effect in reducing the number of possible combinations left. This involves additional data structures and extra processing time.

Now we are beginning to get into Artificial Intelligence, and perhaps beyond the limits of Amstrad Basic; but it is an interesting challenge, and it would give some improvement in performance. If there were grandmasters at this game, the program would become one.

Other ideas you might care to consider are: recording the user's choices and seeing if it can detect, and exploit, colour preference biases in the user; and revising the central data structure by trying out a different, two-dimensional structure. The latter would have four rows for each position in the code and seven columns, one for each colour. Its entries would contain indications of whether each position-colour was possible, impossible, likely or unlikely. Would this enhance the program's performance? We leave you to investigate.

At this point it is time to bid you farewell (and to thank you if you have read through to here rather than skipping directly from Chapter 1). The red, green, blue, cyan, magenta, pink and yellow brick road of games programming stretches out invitingly before you. But you have other avenues to explore as well. We say goodbye with the hope that – partly by reading this book – you are now able to write programs that do something useful, and work.

Appendix A

ASCII codes

The printable ASCII characters are given below, together with their decimal equivalents.

No.	Char.	No.	Char.	No.	Char.
32	[space]	64	@	96	'
33	!	65	A	97	a
34	"	66	B	98	b
35	£	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	–	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_	127	[delete]

The character codes from 128 to 255 are used for block graphics on the Amstrad. For instance, **PRINT CHR$(244)** displays a smiling face and **CHR$(245)** a frowning one. If you want to see what they look like, type in and run the little program below.

```
100  FOR c%=128 TO 254 STEP 2
120     PRINT c%;CHR$(c%),c%+1;CHR$(c%+1)
150  NEXT c%
200  END
```

The characters below 32 are special, non-printing, control characters; but they too can be printed by preceding them with **CHR$(1)**. For example, **PRINT CHR$(1);CHR$(26)** displays a reversed question-mark. Normally, however, they are used for functions like backspacing. The most important are as follows.

CHR$(0) null character, has no effect.
CHR$(1) prefix for printing control codes.
CHR$(2) turns off the text cursor.
CHR$(3) turns the text cursor back on.
CHR$(7) clears the sound queue and sounds the buzzer.
CHR$(8) moves the cursor one space backwards.
CHR$(9) moves the cursor one space forwards.
CHR$(10) moves the cursor one line downwards.
CHR$(11) moves the cursor one line upwards.
CHR$(12) clears the screen.
CHR$(13) carriage return.
CHR$(16) deletes the character under the cursor.
CHR$(24) swaps PEN and PAPER colours ('inverse video').
CHR$(30) moves cursor to the top-left corner.

Appendix B

Basic keywords

This appendix describes Basic's commands and statements – except those for disc file-handling which are listed in Appendix D and pre-defined functions which are dealt with in Appendix F.

In the statement templates, words in capitals (like INPUT) are Basic keywords which should appear as they stand. Words in lower case describe constructions (like 'variable') which should be replaced by a legal example of that construct. Punctuation should appear as shown, except that square brackets [] enclose optional parts and curly braces {} enclose portions that may be repeated zero or more times.

B.1. Basic statements and commands

Statements are normally preceded by a line number in the range 1 to 65535. Multiple statements may be placed on one line, separated by colons.

AFTER waittime [,timernum] GOSUB line
Calls a subroutine after 'waittime' has elapsed, optionally depending on timer number 'timernum' (in the range 0 to 3).

AUTO [line] [,gapsize]
Generates line-numbers automatically for program insertion. The 'line', if present is a constant specifying the starting number; and 'gapsize' is another optional consonant giving the increment between line-numbers. Default for both 'line' and 'gapsize' is 10. If a line generated by AUTO already exists, the system enters EDIT mode.

(AUTO is used as a command, not within a program.)

BORDER hue1 [,hue2]
Sets the colour of the border around the display screen. The parameter 'hue1' must be an expression in the range 0 to 26: see Colour Chart. If the 'hue2' argument is given, it must be in the same range and specifies that the border alternates between the two colours.

CALL address [,arglist]
This calls a machine-code subroutine at memory-location 'address', optionally passing the arguments listed. Enough said!

CLEAR
Clears all variables to zero or the null string, abandons all open files, erases all arrays and user functions, and sets Basic to use radians in trigonometric calculations.

CLEAR INPUT
Discards any characters hanging around in the 'input buffer'.

CLG [inkcode]
Clears the graphics screen. Optionally an 'inkcode' expression in the range 0 to 16 may be given for the graphics background colour.

CLS [# w]
Clears the text screen on window 'w' (an expression in the range 0 to 7). If no window is specified, stream 0 is cleared.

CONT
Attempts to continue execution after a break-in with ESC or after a STOP instruction has been encountered. CONT cannot work if the program has been edited or if it is a protected program.

DATA constant {,constant}
Declares one or more constants for use by READ statements gathered in a program. The constants may be string or numeric, and are all gathered together into one long sequence for reading. (See also RESTORE).

DEF FNname [(arglist)] = formula
Defines a function which can later be invoked by name in Basic expressions to compute a value. The 'name' is like any variable name; the 'arglist' if present is a list of one or more arguments, separated by commas; and the 'formula' is an expression, normally containing the arguments if any, for computing the value of the function. (See Chapter 4.)
(For built-in functions, see Appendix F.)

DEFINT letters
Sets the default data-type for variables to integer. When a variable is encountered without a type marker (% for integer, ! for floating-point, $ for string) at its end, the default type is assumed. Normally the default for all variables is floating-point, but this statement states that variables whose names begin with the 'letters' specified are integers by default. Thus for example

DEFINT a,i–k

specifies that all variable names beginning with 'a' or 'i', 'j' or 'k' are to be treated as integers by default.

DEFREAL letters
This is like DEFINT, except that it specifies the default data-type to be 'real' or floating-point. It is followed by a list of initial letters. Since floating-point is the normal default, the system acts as if the command

DEFREAL a–z

had been given when it starts up.

DEFSTR letters
This is like DEFINT and DEFREAL, only it specifies that variable names with the listed initial letters are to be treated as string variables by default.

DEG
This sets the units for trigonometric computations to be degrees (rather than radians). The initial state of Basic is for the functions COS, SIN, TAN and so on to work with angles measured in radians; but very often it is more convenient to work with degrees. After DEG, Basic uses degrees until a RAD instruction or till one of the commands CLEAR, LOAD, NEW or RUN is issued (or the system is re-booted).

DELETE linespec
This command deletes a line or range of lines from the Basic program in memory. Normal usage is to have a first and last line number separated by a hyphen, as in

DELETE 600–750

which deletes lines 600 to 750 inclusive. But either the first or the last line may be omitted, in which case deletion is from the beginning or to the end of the program. A single line may also be deleted as in

DELETE 800

for example. Be warned that

DELETE

on its own erases the entire program!

DI
This disables all interrupts (other than ESCape) until they are re-enabled by EI or by RETURN from an interrupt-handling subroutine.

DIM name(size) {,name(size)}
This sets up the dimensions of one or more arrays. The 'name' is the array-name, composed according to the normal rules of variable naming.

The 'size' indicates its maximum subscript value. Arrays may have more than one dimension, in which case the upper limits for each subscript value are separated by commas. Variables or expressions may be used to declare an array's size, as well as constants.

Array elements are numbered consecutively from 0 not 1; and the default upper bound for arrays not mentioned in a DIMension statement is 10. (Examples in Chapter 3.)

DRAW ac,up [,[inkcode] [,inkmode]]
This draws a line on the graphics screen from the current graphics cursor position to the position given by 'ac' (horizontal displacement 0 to 640) and 'up' (vertical displacement 0 to 400). An 'inkcode' in the range 0 to 15 may be specified giving the colour of the line. The optional 'inkmode' determines how the ink interacts with the ink already on the graphics screen. The four ink modes are: 0 normal default (overwriting), 1 XOR with existing colour, 2 AND with existing colour, 3 OR with existing colour.

All parameters are numeric expressions.

DRAWR ac,up [,[inkcode] [,inkmode]]
This is the relative version of DRAW. It draws a line in the same way except that the destination point is given relative to the current graphics cursor position. Thus the expressions 'ac' and 'up' are not absolute addresses on the screen but offsets relative to the latest position plotted.

EDIT line
This command enters editing mode on the 'line' number specified (which must be a constant, not a variable). The arrowed cursor keys, with DEL and CLR, may be used to amend that line. (See Chapter 2.)

EI
This enables interrupts that have been turned off by DI. If interrupts are disabled in a subroutine, they are automatically re-enabled by the RETURN instruction which terminates execution of that routine.

END
Ends the execution of a program. An END is automatically assumed after the final line of the program, if none is present.

ENT envelope {,stepnum,stepsize,stepgap}
This defines a tone envelope for reference in a SOUND instruction elsewhere in the program, with an identifying number 'envelope' in the range 1 to 15. The numeric parameters 'stepnum', 'stepsize' and 'stepgap' give the number of steps (0 to 239), the pitch increment or decrement (−128 to +127) and the time between steps (0 to 255) respectively. Up to five groups of three parameters may follow the 'envelope' number. They specify

how fast and how far the basic pitch is to be varied during the SOUND output. (See Chapter 8.)

ENV envelope {,stepnum,stepsize,stepgap}
This defines a volume envelope for use by reference to the 'envelope' number (1 to 15) in a SOUND command. The format is the same as for a tone envelope, above, except that the 'stepsize' (−128 to +127) refers to loudness rather than pitch. The volume range is from 0 (quiet) to 15 (full volume) but it recycles to zero if incremented beyond 15. The maximum number of parameters to the ENV command is 16 – one identifying number plus 5 sections of three parameters.

ERASE variable {,variable}
This erases the contents of a variable (normally an array-name) when it is no longer required, releasing the storage it occupied for other uses. Thus the statement

1250 ERASE Zonk, Junk$

removes all traces of a numeric array called **Zonk** and a string array called **Junk$**.

ERL
This is a system variable that reports the line number of the last error detected by Basic. It can be used with ERR and ON ERROR GOTO to handle errors under program control. (See also Appendix E.)

ERR
This is another special system variable that is set to contain the error-code of the last error detected by Basic. For a list of error numbers, see Appendix E.

ERROR numeric
This forces Basic to act as if the error designated by the integer 'numeric' expression had occurred. If an

ON ERROR GOTO line

has been set up that 'line' will be executed with the values of ERR and ERL set appropriately.

EVERY waittime [,timernum] GOSUB line
This calls the subroutine beginning at line-number 'line' every so often. The optional 'timernum' (0 to 3) specifies one of the four interrupt timers, and the 'waittime' expression gives the interval in fiftieths of a second. Timer 0 is used if none is specified. Timer 3 has the highest priority. Each timer may have a routine associated with it.

FILL inkcode
This fills an area of the graphics screen with the hue given by the 'inkcode'

expression (in the range 0 to 15). Filling begins at the current graphics cursor location and continues till a boundary is reached. A boundary may be: (1) the edge of the screen; (2) a line in the current graphics pen colour; or (3) a line in the ink being used for FILLing.

FOR counter = firstval TO lastval [STEP stepsize]
Opens a loop controlled by a 'counter' variable. The numeric formula 'firstval' is evaluated and assigned to the index 'counter'. Then the instructions up to the NEXT with the same counter variable are executed (provided that the counter is not already greater than the value of the numeric expression 'lastval'). Then the counter is incremented by the value of the expression after STEP (or +1 if this part is absent) and control returns to the statement immediately after the FOR. This goes on until the counter exceeds the final value 'lastval', when the program carries on with the statement after NEXT.

If the optional 'stepsize' is negative, looping continues till the counter is less than the last value (not greater).

FRAME
Synchronizes the writing of graphics with the frame flyback of the screen. The effect is that movement on the screen appears smoother, with less flickering.

GOSUB line
Transfers execution to the statement with the line number specified. Execution then continues normally until a RETURN is reached, when control returns to the instruction just after the GOSUB. Subroutines in Amstrad Basic may be recursive, but there is no parameter-passing mechanism. (See Chapters 4 and 9.)

The 'line' must be a constant, not a variable or expression.

GOTO line
Branches to the line number given and continues execution from there. The 'line' must be a constant between 1 and 65535.

GRAPHICS PAPER inkcode
Sets the background colour to 'inkcode', an expression in the range 0 to 15 specifying one of the sixteen ink hues. In mode 1 the inks are 'folded' so that four colours are repeated four times. In mode 2 two colours are repeated eight times. (See also inside flap.)

GRAPHICS PEN inkcode [,modecode]
This sets the graphics foreground colour, used for plotting and drawing lines, to that specified by the expression 'inkcode' (0 to 15). Optionally the 'modecode' parameter may be given a value of 0 for an opaque background or 1 for a transparent background. (This affects dotted lines and so on.)

IF truthval THEN statements [ELSE statements]
If the 'truthval' expression yields a non-zero value (true) the statement or statements after THEN are executed. If it has a value of zero (false), the statement or statements after ELSE are executed if present, otherwise nothing happens. The 'truthval' is usually a logical expression using logical and/or comparison operators. (See Chapter 2 and Section B2.)

INK inkcode,huecode [,flasher]
This assigns the colour indicated by the expression 'huecode' (0 to 26) to the ink number (0 to 15) given by the expression 'inkcode' for use in subsequent PEN and PAPER instructions. If the 'flasher' expression is also given, it must also be in the range 0 to 26, and indicates that the colours 'huecode' and 'flasher' are to alternate. See Colour Chart and Ink Codes on back flap for details.

INPUT [#chan,] [;] [message punc] variable {,variable}
This accepts data input from the stated channel 'chan' or stream 0 if not specified. The values typed by the user are given, in order, to the variables listed. Commas are used to separate data items on input. Strings for input need not be in quotes unless they contain commas or leading or trailing blanks.

The first, optional, semicolon [;] suppresses the new line that otherwise occurs after the INPUT is executed. The 'message' is a quoted string which may optionally be used as a prompt. The 'punc' following the optional prompt-string must be a comma or a semicolon: the latter causes a question mark to be displayed; the former suppresses it.

If the user gives the wrong number or wrong type of data, Basic responds with the error message.

?Redo from start

and the user must re-enter the whole input list.

KEY keytoken, stringex
Assigns the string expression 'stringex' to a key designated by the integer expression 'keytoken'. Key tokens 0 to 31 refer to the key values 128 to 159.

This allows instruction sequences to be stored and called up by a single key-stroke. Key 0 (code 128) for example is f0 on the function key-pad. Normally it just contains a "0", but by giving the command

```
KEY 0, "LOAD"+CHR$(34)+"PROG"+CHR$(34)+CHR$(13)+"GOSUB 1000"+
CHR$(13)
```

you could set the machine up so that merely pressing function key 0 would have the same effect as typing the two lines

LOAD "PROG"
GOSUB 1000

in full. (Function keys 0 to 9 have key tokens 0 to 9.)

KEY DEF keynumb, rept [,norm [,shifted [,control]]]
An instruction so mind-bogglingly arcane that several years of study in a Himalayan monastery are necessary even to formulate its parameters correctly. (They say that what you leave out of a programming language is even more important than what you put in.)

[LET] variable = formula
Assigns the value of the formula or expression on the right to the variable or array-element on the left. The LET keyword is optional. Numeric variables must be given numeric values, and string variables must be given string values. If an integer variable is assigned a fractional value, the result is rounded to the nearest whole number.

LINE INPUT [#chan,] [;] [message punc] variable
Gets a complete line of input from the keyboard (including commas and spaces etc.) and assigns it to the string 'variable' concerned. The first, optional, semicolon [;] suppresses the newline at the end of the input.

The 'message' is a quoted string constant used as a prompt, and the 'punc' is either a comma or semicolon – the former being used when no question mark is to be displayed after the prompt-string.

Input from the user is terminated by pressing the RETURN key, or when 255 characters are received, whichever comes sooner.

The 'chan' parameter is 0 by default (normal keyboard input). If 'chan' is equal to 9 then the current disc (or tape) file is used for input.

LIST [line] [–line] [,# chan]
Lists a program line or series of lines. LIST on its own lists the entire program. If 'chan' is not given, stream 0 is used (display screen). If 'chan' has a value 1 to 7, then the appropriate WINDOW is used for listing. If 'chan' equals 8, the printer is used for hard-copy listing.

LOCATE [# chan,] ac,down
Positions the text cursor at the position specified – 'ac' character positions across to the right and 'down' lines downwards, where 'ac' and 'down' are numeric expressions. In mode 0, the screen width is 20 positions; in mode 1 it is 40 positions; and in mode 2 it is 80. The depth is always 25.

If a 'chan' expression is given, the position is within the WINDOW specified (1 to 7). LOCATE 1,1 puts the text cursor at the top left.

MEMORY address
Allocates the amount of RAM available to Basic by setting the 'address' of the

highest byte. Only useful if you want to hide machine-code routines in a section of memory that Basic cannot reach.

MID$(variable,position [,newsize]) = stringex
Inserts the string expression 'stringex' into the string 'variable' at the 'position' given. The number of characters transferred may be specified by the integer expression 'newsize'. Thus

MID$(Text$,p%,4) = "Helper"

puts the 4 characters "Help" into the string variable **Text$** starting at position **p%** within the receiving string variable. (See also the MID$() function in Appendix F.)

MODE integer
This clears the screen and changes to mode 0, 1 or 2 – as given by the 'integer' expression. (See Chapter 8.)

MOVE ac,up [,[inkcode] [,inkmode]]
Moves the graphics cursor to the location specified by the horizontal and vertical coordinates 'ac' and 'up' (0 to 640 and 0 to 400 respectively). It may optionally select a new graphics pen with the 'inkcode' expression (0 to 15). The optional 'inkmode' parameter is as for DRAW.

MOVER ac,up [,[inkcode] [,inkmode]]
This instruction moves the graphics cursor to a position specified by the numeric expressions 'ac' and 'up' relative to the current graphics cursor position. It is the relative version of MOVE, above, and the parameters other than 'ac' and 'up' have the same meanings.

NEW
This command clears memory in preparation for the insertion of a new program.

ON BREAK CONT
This prevents the action of the ESC key from halting the program. ESC is disabled until the next ON BREAK STOP instruction, so take care with this one.

ON BREAK GOSUB line
Sets up an interrupt-handling routine. When next ESC is pressed twice, the program will not halt but will transfer to the subroutine at the 'line' specified.

ON BREAK STOP
This cancels the effect of an earlier ON BREAK CONT command and restores BASIC to its normal mode of handling break-ins with the ESCape key.

ON ERROR GOTO line
Changes Basic's normal mode of error-handling so that, instead of stopping with an error diagnostic, control is transferred to the line number specified when the next error occurs. Having dealt with the error, RESUME can be used to carry on where the program left off. If the error cannot be handled, ON ERROR GOTO 0 re-instates Basic's normal way of processing errors.
(See also Appendix E.)

ON selector GOSUB line {,line}
Picks a subroutine to jump to according to the value of the integer expression 'selector'. If the value is 1 the first 'line' is chosen; if it equals 2, the second is chosen, and so on. If the selector is less than 1 or more than the number of lines after the GOSUB, no subroutine will be called and control will 'fall through' to the next statement.

ON selector GOTO line {,line}
Transfers to one of a series of line numbers as with ON/GOSUB above, but by a GOTO rather than a GOSUB.

ON SQ(chan) GOSUB line
Goes to a Basic subroutine when there is a free time-slot in the sound queue specified. The 'chan' parameter should be an integer expressioin yielding the value 1 for channel A, 2 for channel B or 4 for channel C.

ORIGIN wide,high [,east,west,up,down]
Re-sets the graphics origin (zero,zero coordinate) to the position specified by the expressions 'wide' and 'high'. This instruction may also be used to define a graphics window by means of the four optional parameters, which set new boundaries for the graphics area. (Plotting beyond the physical edge of the screen never has any effect.)

OUT port,integer
Sends the 'integer' value (0 to 255) to the specified 'port' or I/O device. (Only for machine-code freaks.)

PAPER [# chan,] inkcode
PAPER selects the background colour for text. The 'inkcode' expression (range from 0 to 15) picks one of the available hues. The 'chan' specifies a window (0 to 7) and defaults to 0 if not given. (See back flap.)

PEN [# chan,] inkcode [,modecode]
Sets the 'inkcode' (0 to 15) to be used when writing text to the given stream ('chan' 0 to 7, 0 if not specified). The optional 'modecode' parameter can be used to set background mode to opaque (0) or transparent (1). See also back flap.

PLOT ac,up [, [inkcode][,inkmode]]
Plots a point on the graphics screen at the position given by the 'ac' and 'up' coordinates (ranges 0 to 640 and 0 to 400). The 'inkcode' expression (0 to 15) may be used to select the colour of the point. The 'inkmode' (0 to 4) determines how the ink interacts with those already on the screen, as with the DRAW command.

PLOTR ac,up [,[inkcode] [,inkmode]]
The relative version of PLOT is the same as above except that the expressions 'ac' and 'up' are interpreted as offsets from the current graphics cursor position.

POKE address,integer
Puts the value of the 'integer' expression (0 to 255) directly into the memory location given by 'address'.

PRINT [#chan,] [item] {punc item}
Prints a list of values to the output channel given by 'chan' (0 to 9). If no 'chan' is specified, stream 0, the screen, is used. A question mark (?) may be used as a synonym for the keyword PRINT.

Each 'item' may be a constant, variable or expression. The separators 'punc' may be commas for zoned spacing or semicolons for closely-packed spacing. The pseudo-functions SPC(n) and TAB(n) may be used as items in the output list. SPC(8), for example, causes 8 spaces to be printed; while TAB(8) causes the cursor or print-head to move to the 8th character position. (If it has already gone beyond that position, it will move forwards to that position on the next line.)

Channel 8 is for printed output, if a printer is connected; and channel 9 is for disc-file output, if a disc is installed.

PRINT [# chan,] USING form ; item {,item}
This prints a list of expressions on the specified channel (or stream 0, the screen display, if none is given) whose layout is governed by the format string 'form'. Details of PRINT USING formatting are given in Chapter 6.

RAD
Causes radians to be used as the units for trigonometric calculations, e.g. in SIN() and COS(). This is the default mode, so RAD is only needed after a DEG instruction has altered the trigonometric units to be degrees.

RANDOMIZE numeric
Resets the seed of the pseudo-random number generator to the value given by the 'numeric' expression. This prevents the RND function from repeating the same series of values. RANDOMIZE TIME is a handy way of randomizing the generator.

READ variable {,variable}
Extracts constants from DATA statements into a list of variables and/or array-elements. String constants must be read into string variables and numeric values into numeric variables. The first READ gets the first DATA item in the program, and subsequent READs carry on where previous ones left off. A RESTORE instruction may be used to return to the start of the data area.

RELEASE sq
Releases sound channels that have been set to hold in a SOUND command. The 'sq' parameter is an integer in the range 1 to 7 which specifies a group of sound channels by adding up their individual code numbers (1 for A, 2 for B and 4 for C).

REM textline
Introduces a non-executable comment 'textline', which may be any printable characters. If execution is sent to a REM statement it has no effect: control passes on to the next statement. Remarks may also be appended to any line except a DATA statement by typing an apostrophe ('), which marks all characters following up to the end of the line as commentary.

RENUM [newline] [,oldline] [gapsize]
This command renumbers program lines. The parameters are line number (constants in the range 0 to 65535). If 'oldline' is omitted, renumbering commences from the beginning of the program. If 'newline' is omitted, the renumbered program will start from line 10. The 'gapsize' specifies the step between consecutive line numbers: if omitted, the lines will go up in steps of 10.

RENUM takes care of all GOTO, GOSUB and other destinations during resequencing. However, line references within string expressions – such as in KEY commands – are not altered.

RESTORE [line]
Resets the data pointer back to the first item of the first DATA line in the program. If an optional 'line' is given, the pointer is reset to the line number specified (which must be a constant).

RESUME [line]
Resumes normal execution after an error has been trapped by an ON ERROR GOTO and processed. If no 'line' is specified, resumption is at the line where the error occurred. RESUME NEXT has the effect of re-starting at the line after the one where the error occurred.

RETURN
Returns from a subroutine by sending control back to the instruction just after the most recently executed GOSUB.

RUN [line]
Executes the Basic program currently in memory. If a 'line' is given, execution commences from the line number specified.

SOUND chan, tone [,duration [,loudness[,ve[,te[,np]]]]]
Sounds the bleeper on 'chan' (1, 2 or 4) with the 'tone' specified for 'duration' hundredths of a second at volume level 'loudness' (0 to 15). Among the optional parameters, 've' specifies a volume envelope (see ENV), 'te' specifies a tone envelope (see ENT) and 'np' (0 to 31) adds noise of varying pitch. (See Chapter 8 for details.)

SPEED INK period1, period2
Sets the rate of alternation for two colours specified in a BORDER or INK command. The numeric expression 'period1' specifies the time in fiftieths of a second for the first colour; 'period2' sets the time in fiftieths for the second colour.

WARNING: flicker frequencies in tune with brain rhythms can have dangerous effects, especially on people susceptible to epilepsy.

SPEED KEY wait, rate
Sets the timing for auto-repeat on keyboard keys. The expression 'wait' gives the time a key must be depressed, in fiftieths of a second, before auto-repeating begins. The expression 'rate' gives the interval between repetitions, again in fiftieths. The normal settings are 30 and 2. (Small values of 'wait' are best avoided.)

STOP
Stops a program running, but leaves it in a state where it can be resumed by the CONT command – provided it has not been edited in the meantime.

SYMBOL charnum, rowlist
Redefines the shape of a character for screen display. The 'charnum' is an ASCII code in the range 0 to 255 (see Appendix A), normally greater than 127. The 'rowlist' contains eight integers in the range 0 to 255, separated by commas: these define a row of light and dark dots according to their bit-pattern. Thus, for instance, 37 (00100101 binary) denotes a line of off-off-on-off-off-on-off-on as one of the eight rows making up a character matrix.

(Examples can be seen in Listing 8.4.)

SYMBOL AFTER integer
This sets the first character in the ASCII sequence that may be redefined by a SYMBOL command. The 'integer' is an expression in the range 0 to 256, which selects the lowest-numbered character that may be redefined. Thus SYMBOL AFTER 200 means that all ASCII codes from 200 upwards may be

redefined. The initial setting is for SYMBOL AFTER 240, giving 16 user-defined characters, numbered 240 to 255.

TAG [# chan]
Sends any text for the stream given by 'chan' (0 to 7) to be printed at the graphics cursor position (instead of the text cursor position). If no channel number is specified, stream 0 – the screen display – is used.

TAGOFF [# chan]
Cancels the effect of a prior TAG instruction for the given window or stream, or for the main display screen if none is given.

TROFF
Turns off line-number tracing set up by TRON.

TRON
Traces execution of program lines by printing each line number in square brackets before executing it, until cancelled by TROFF. Used sparingly, this can be helpful during debugging.

WHILE truthval
Opens a loop, closed by a following WEND, which will be executed repeatedly as long as the logical expression 'truthval' is true (i.e. has a non-zero value). See Chapter 3.

WIDTH integer
Informs Basic how many characters fit onto one line of printer output (on channel 8). A width of 132 is assumed unless a WIDTH command overrides it.

The instruction WIDTH 255 tells Basic not to insert extra carriage-return/line-feeds whatever the length of an output line: it is then up to the printer to put in line-breaks if necessary.

WINDOW [#chan,] east,west,up,down
Declares the dimensions of a text window on the screen. The parameters specify the left, right, upper and lower limits in character positions (consistent with the MODE in use).

If no 'chan' (0 to 7) is given, stream 0, the main screen, is assumed.

WINDOW SWAP window1, window2
Exchanges the text of two specified windows completely. (Note there is no hash-sign (#) in front of either parameter.)

ZONE integer
Changes the width of the print zones used by the PRINT statement when commas are used to delimit items. The default setting is ZONE 13, but the 'integer' expression may take any value from 1 to 255.

B.2. Basic operators

Operators are used to link operands (constants, variables or subexpressions) in expressions.

Logical operators

AND Bitwise Boolean AND on integers: result bit is 0 unless both left and right operand bits are 1.

NOT Bitwise negation: inverts every bit in its single operand.

OR Bitwise logical inclusive OR: result in each bit position is 1 unless both operand bits are 0.

XOR Bitwise exclusive OR: result bit is 1 if the two operand bits differ, 0 if they are the same.

Relational Operators

< Less than: returns −1 (true) if left operand is numerically smaller than right operand, otherwise 0 (false).

> Greater than: returns true if left argument is numerically greater than right operand, otherwise false.

= Equal to: returns true if both operands are numerically equal, otherwise false.

<> Not equal: returns true if both operands differ, false if they have equal values.

<= Lesser or equal: returns true if left operand is less than or equal to right operand, otherwise false.

>= Greater or equal: returns true if left operand exceeds or equals right operand, otherwise false.

Arithmetic operators

+ Addition: returns the sum of its two operands.

− Subtraction: returns the difference between its two operands.

* Multiplication: returns the product of its two operands.

/ Division: returns the ratio of the left operand divided by the right operand.

\ Integer division: returns the whole-number result (truncated towards zero) of dividing the left by the right operand.

MOD Modulus: remainder after dividing left operand by right operand.

^ Exponentiation: result of raising left operand to the power of right operand.

Appendix C

CP/M Commands

One of the joys of the Amstrad CPC 664 and even more the CPC 6128 is that it permits the use of the CP/M operating system. CP/M is not the world's best operating system (though there are many worse) but it has been around so long that a cornucopia of valuable software has accumulated for it.

It cannot be covered in a mere Appendix; but here at least is a list of its built-in commands to get you started. The ones that require filename parameters use the same naming conventions as AMSDOS – i.e. up to eight characters in a name, then up to three characters after a dot in the extension. You can use the ? and * 'wild-cards' too. (See Chapter 7.)

A:
Makes A: the current disc drive. (Default.)

B:
Makes B: the current disc drive (if you have two).

DIR [filespec]
Gives a directory of the current disc. The 'filespec' may be used to list groups of files, e.g.

DIR *.COM

lists all files with a .COM extension.

ERA filespec
Erases a file or group of files from the directory.

REN newspec = oldspec
Renames the file named by 'oldspec' with the name given by 'newspec'.
For example

REN WEAK.TEA = MELTING.ICE

renames the file currently called MELTING.ICE as WEAK.TEA (Wild cards may not be used with REN.)

TYPE filespec
Types the specified file or files on the screen. If CONTROL/P is in effect, the output will also be copied to the printer.

USER n
Changes the current user number to *n*, where *n* is from 0 to 15. CP/M starts with user zero selected; but by altering the number you can set up several groups of related files on one disc. (With less than 175K to spare! Who do they think they are kidding?)

There are also a variety of useful control characters which do not all function in the same way under CP/M as they do in AMSDOS. The main ones are listed below.

CONTROL/C	Abandon operation (like ESCape in Basic).
CONTROL/G	Delete character under the cursor.
CONTROL/H	Backspace and delete.
CONTROL/I	Tab to next tab stop.
CONTROL/K	Delete to end of line.
CONTROL/P	Turn on copying screen output to printer (or turn it off if already on).
CONTROL/Q	Resume screen listing frozen by CONTROL/S.
CONTROL/R	Re-echo the current command line as CP/M sees it (useful after multiple deletions).
CONTROL/S	Freeze screen output (till another CONTROL/S).
CONTROL/U	Erase the current line.
CONTROL/Z	End-of-file or end-of-input marker.

The DEL and RETURN keys act as they do in Basic or under AMSDOS. The arrowed cursor keys can also be used under CP/M.

To run a compiled program under CP/M, e.g. a utility saved on disc with .COM extension, just give its name. Thus

DISCKIT3

runs DISCKIT3.COM from disc (for formatting, copying etc.).

Appendix D

Disc-file Instructions

The statements and functions for file-handling on disc have been grouped together here for convenience. (For further details on datafile handling, and a list of AMSDOS commands, see Chapter 7.)

CAT
This command displays a disc catalogue, giving names and extensions of files with their lengths in kilobytes.

CHAIN filename [,line]
Loads a program named by the string expression 'filename' into memory and runs it. Execution normally starts with the first program line, but the parameter 'line' may be used to specify a different starting point.

CHAIN MERGE filename [,line] [,DELETE linespec]
Loads and runs a program from disc, merging it into the current program. The optional 'DELETE linespec' section may be used to delete a range of lines in the current program before loading the new one. Note that lines in the present program will be overwritten by lines with the same number in the incoming one.

CLOSEIN
Closes any disc input file.

CLOSEOUT
Closes any output file opened on disc channel 9.

DERR
A system variable used by AMSDOS to give details of any disc error trapped by ON ERROR GOTO. (See Appendix E.)

EOF
This function tests whether the disc input file has any more input data. It returns −1 (true) if the file is exhausted, otherwise 0 (false).

INPUT #9, item {,item}
The form of the INPUT instruction used for reading from disc files. See Appendix B.

LOAD filename [,address]
Loads a file named by the string expression 'filename' into memory, replacing any existing program. Specifying the optional 'address' will load a binary machine-code file at the address given rather than the one it was saved from.

MERGE filename
Merges the program named by the string expression 'filename' into working memory on a line-by-line basis, as if the incoming program lines had been typed at the keyboard. This is useful for appending subroutine libraries saved on disc.

OPENIN filename
This statement opens the file named by the string expression 'filename' on disc channel 9 – for input. It must be an ASCII file.

OPENOUT filename
This statement opens an output file named by the string expression 'filename' on disc channel 9. Both OPENIN and OPENOUT work with ASCII sequential files only.

RUN filename
Runs a Basic or machine-code program saved on disc under the name 'filename' (which is a string expression). Any previous program is cleared from memory.

SAVE filename [,filetype]
Saves the current program in working memory to disc with the name given by the string expression 'filename'. If the 'filetype' is omitted, it is saved as a normal Basic program with .BAS extension if none is specified. A back-up of any existing file with the same name will be made just prior to saving, with .BAK extension.

If 'filetype' is P, the program will be saved in protected mode. This means that you can run but not list it.

If the filetype is A, the program will be saved in plain ASCII format. This means that it can be read by a Basic program using OPENIN and INPUT#9 like any other sequential text file. This is handy for creating simple 'programs' which are really sequential datafiles, or for analysing Basic programs.

WRITE [# chan,] item {,item}
Writes the various 'items' of data to a file (assuming 'chan' has a value of 9) in a form suitable for re-reading by a Basic program. Output items are separated by commas and strings are enclosed by double-quotes.

Appendix E

Error messages

E.1. Basic errors

The error-numbers given below with the Basic error messages are assigned to the special variable ERR when an error occurs. This allows trapping of errors (using the ON ERROR GOTO command) in Basic programs. Likewise the special variable ERL is set to the line number where an error occurred by Basic. Your programs can use this information to deal with various error conditions without halting execution.

1 Unexpected NEXT
A NEXT command has been encountered while not in a FOR loop, or the control variable in the NEXT command does not match that in the FOR.

2 Syntax Error
Basic cannot understand the given line because a construct within it is not legal.

3 Unexpected RETURN
A RETURN instruction has been encountered when not in a subroutine.

4 DATA exhausted
A READ instruction has attempted to read beyond the end of the last DATA.

5 Improper argument
This is a general purpose error. The value of a function's argument, or a command parameter is invalid in some way.

6 Overflow
The result of an arithmetic operation has overflowed. This may be a floating point overflow, in which case some operation has yielded a value greater than 1.7E↑38 (approximately). Alternatively, this may be the result of a failed attempt to change a floating point number to a 16-bit signed integer.

7 Memory full
The current program or its variables may be simply too big, or the control structure is too deeply nested (nested GOSUBs, WHILEs or FORs).

A MEMORY command will give this error if an attempt is made to set the

top of Basic's memory too low, or to an impossibly high value. Note that an open file has a buffer allocated to it, and that may restrict the values that MEMORY may use.

8 Line does not exist.
The line referenced cannot be found.

9 Subscript out of range
One of the subscripts in an array reference is too large or less than zero.

10 Array already dimensioned
One of the arrays in a DIM statement has already been declared.

11 Division by zero
May occur in real division, integer division, integer modulus or in exponentiation.

12 Invalid direct command
The last command attempted is not valid in direct mode.

13 Type mismatch
A numeric value has been presented where a string value is required or vice versa, or an invalidly formed number has been found in READ or INPUT.

14 String space full
So many strings have been created that there is no further room available, even after 'garbage collection'.

15 String too long
String exceeds 255 characters in length. May be generated by appending strings together.

16 String expression too complex
String expressions may generate a number of intermediate string values. When the number of these values exceeds a system-imposed limit, this error results.

17 Cannot CONTinue
For one reason or another the current program cannot be restarted using CONT. Note that CONT is intended for restarting after a STOP command, **[ESC][ESC]**, or an error, and that any alteration of the program in the meantime makes a restart impossible.

18 Unknown user function
No DEF FN has been executed to define the FN just invoked.

19 RESUME missing
The end of the program has been encountered while in error processing mode (i.e. in an ON ERROR GOTO routine).

20 Unexpected RESUME
RESUME is only valid while in error processing mode (i.e. in an ON ERROR GOTO routine).

21 Direct command found
When loading a file, a line without a line number has been found.

22 Operand missing
Basic has encountered an incomplete expression.

23 Line too long
A line when converted to Basic's internal form becomes too big.

24 EOF met
An attempt has been made to read past the end of the file on the input stream.

25 File type error
The file being read is not of a suitable type. OPENIN is only prepared to open ASCII text files. Similarly, LOAD, RUN, etc., are only prepared to deal with file types produced by SAVE.

26 NEXT missing
Cannot find a NEXT to match a FOR command. A line number accompanying this message indicates the FOR command to which this error applies.

27 File already open
An OPENIN or OPENOUT command has been executed before the previously opened file has been closed.

28 Unknown command
Basic cannot find a taker for an external command, i.e. a command preceded by a bar |.

29 WEND missing
Cannot find a WEND to match a WHILE command.

30 Unexpected WEND
Encountered a WEND when not in a WHILE loop, or a WEND that does not match the current WHILE loop.

31 File not open
See Section E.2.

32 Broken in
See Section E.2.

E.2. AMSDOS errors

All disc errors put the number 32 into ERR, but they can be distinguished by looking at the state of variable DERR, which returns values as follows.

0 ESC has been pressed.
22 ESC has been pressed.
142 The data stream is not in a suitable state.
143 Hardware end-of-file has been reached.
144 Bad command (e.g. incorrect file name).
145 File already exists.
146 File does not exist.
147 Disc directory is full.
148 Disc space is full up.
149 Disc has been exchanged while files were open.
150 File is Read-Only.
151 Software end-of-file has been detected.

Appendix F

Pre-defined functions

Basic provides a large number of functions to supplement the arithmetic operators and for other purposes. In the explanations below **x,y,i,n,s$,ss$** are used as follows.

x, y	numeric expressions,
s$, ss$	string expressions,
i, n	numeric expressions with integer values.

ABS(x)	returns the absolute value of its argument, ignoring sign.
ASC(s$)	returns the numeric ASCII code of the first characters in its argument.
ATN(x)	calculates the arc-tangent of its argument.
BIN$(i,n)	produces a string of binary digits representing the value of its first argument, to the length specified by the second argument (0 to 16).
CHR$(n)	converts an integer expression in the range 0 to 255 to its equivalent as a one-character string.
CINT(x)	rounds its numeric argument to the nearest integer.
COPYCHR$(#n)	copies a character from the current text cursor position in window **n**, which must be 0 to 7.
COS(x)	calculates the cosine of its argument.
CREAL(x)	converts its argument to a 'real', i.e. floating-point, number.
DEC$(x,form$)	returns a decimal string representation of the numeric argument using the **form$** string argument as a format template in the same way as a PRINT USING statement.
EXP(x)	raises *e* to the power given, where *e* is 2.71828 approximately – the base of natural logarithms.
FIX(x)	rounds its argument towards zero, giving the nearest whole number but as a floating-point result (not an integer like CINT).
FRE(0)	returns the amount of free memory unused, in bytes.
FRE("")	returns the amount of free memory, like **FRE(0)**, but performs a 'garbage collection' first.
HEX$(i,n)	produces a digit-string giving the hexadecimal (base 16) representation of the first argument, with length given by the second argument.

HIMEM returns the highest byte address usable by Basic: may be affected by the MEMORY command.

INKEY(i) interrogates the keyboard as explained in Chapter 6.

INKEY$ returns a one-character string reflecting which key is pressed, or the null string if no key is currently being pressed.

INP(n) gives the status of hardware port number *n*.

INSTR(s$,ss$) searches for the second string within the first one and returns the position where it was found, or 0 if it was not found.

INSTR(i,s$,ss$) as above, but starting the search at position **i** in **s$**.

INT(x) truncates the numeric argument to the nearest smaller whole number.

JOY(n) reads the joystick status (if you have one) where **n** is 0 or 1.

LEFT$(s$,n) returns a string containing the leftmost **n** characters of the string **s$**.

LEN(s$) returns the number of characters in **s$**.

LOG(x) computes the natural logarithm (base *e*) of **x**, which must exceed zero.

LOG10(x) computes the base-ten logarithm of its argument, which must be greater than zero.

LOWER$(s$) returns a string the same as **s$** except that any letters have been forced into lower-case.

MAX(arglist) returns the largest value in 'arglist' which is a series of numeric expressions separated by commas.

MID$(s$,i,n) returns a new string being **n** characters long, taken from position **i** onwards in **s$**: if the third argument is absent, the rest of the string from position **i** is returned.

MIN(arglist) returns the smallest value in a list of expressions.

PEEK(n) looks at the given memory address and returns its value (0 to 255).

PI returns an approximation to Pi, the ratio between circumference and diameter of a circle.

POS(#n) reports the present horizontal position of the text cursor in window number **n**, which must be specified (0 to 7).

REMAIN(n) returns the time remaining on delay timer **n** (0 to 3) and disables that timer.

RIGHT$(s$,n) returns the last **n** characters from string **s$**.

RND yields the next fraction, in the range 0 to 1, from the built-in random number generator.

ROUND(x,n) rounds its first argument to **n** decimal places.

SGN(x) determines the sign: −1 if **x** is negative, 0 if it is zero, +1 if **x** is positive.

SIN(x) computes the sine of its numeric argument.

SPACE$(n) returns a string of **n** space characters.

SQ(n) reports the status of sound queue **n**.

SQR(x)	returns the positive square root of a non-negative number, but produces an error if its argument is negative.
STR$(n)	returns the string representation of a number.
STRING$(n,s$)	returns a string containing **n** copies of the argument **s$**.
TAN(x)	calculates the tangent of its numeric argument, which must lie in the range −200000 to 200000.
TEST(x,y)	moves the graphics cursor to coordinate **x,y** and returns the ink-number of the hue it finds there.
TESTR(x,y)	moves the graphics cursor by the horizontal and vertical displacements **x** and **y**, returning the ink-code it finds at that point.
TIME	gives the elapsed time since the computer was switched on or reset, in 300ths of a second.
UNT(n)	returns an integer in the range −32768 to +32767 which is the two's-complement equivalent of the unsigned value of its argument (0 to 65535).
UPPER$(s$)	returns a string with the same characters as **s$**, except that all letters are in upper case.
VAL(s$)	converts a string representation of a number into its numeric equivalent.
VPOS(#n)	reports the vertical position of the text cursor in window **n**, which must be in the range 0 to 7.
XPOS	returns the current horizontal position to the graphics cursor.
YPOS	returns the current vertical position of the graphics cursor.

Appendix G

Glossary of computing terms

ADDRESS The location in a computer's memory where a number or instruction is stored.

ALGORITHM A method of solving a problem (often mathematical) in a finite number of steps.

ALPHANUMERIC Composed of letters and/or digits only.

AMSDOS The Amstrad Disc Operating System, a program that controls all disc-access from within Basic.

ARGUMENT Variable passed to a subroutine or function when it is called. (See also Parameter.)

ARRAY Collection of data under one name whose members are individually identified by subscripts.

ARTIFICIAL INTELLIGENCE The study of ways of making machines (especially computers) solve problems in an intelligent fashion, typically by behaving more like human beings than machines usually do.

ASCII American Standard Code for Information Interchange.

BASIC Beginner's All-purpose Symbolic Instruction Code.

BINARY Related to the number system based on 2, using only the digits 0 and 1.

BIT Binary digit – either 0 or 1 – the elementary unit of information.

BOOLEAN Referring to the logical algebra developed by George Boole in the 19th century where propositions may be true or false only. True and false are usually signified as 1 and 0.

BOOTSTRAP System programs such as CP/M do not load themselves into RAM. Normally they are 'bootstrapped' into memory by a small routine resident in ROM that initiates the loading process at a specific memory location.

BUFFER An old fool. Alternatively a queue in RAM where pending information is held before it is processed.

BUG A fault in a program that causes it to fail, hence 'de-bugging'.

BYTE A group of 8 bits.

CALL Transfer control to a subroutine or function.

CARRIAGE-RETURN The movement of a print-head or display cursor to the start of a new line.

CHAINING Splitting up a large program into smaller linked segments.

CHANNEL A route for computer output to a peripheral, particularly disc. See Stream.

CHARACTER A single symbol that a computer can recognize, e.g. letters A–Z, numerals 0–9 and various other signs.

COLD BOOT Re-starting a computer system from scratch.

COMMAND An instruction that causes Basic to take immediate action, generally issued outside a program.

COMPILER A program that translates from a programming language such as Basic into the computer's own instruction code.

COMPUTER Machine for processing information according to a program of instructions.

CONSTANT Fixed value.

CPU Central Processing Unit.

CP/M Control Program/Monitor. An operating system written by Gary Kildall of Digital Research for small computers based on the 8080 or Z80 chips.

CURSOR A movable marker indicating where on the screen text or graphics is to appear.

DATA Information available to a program.

DATABASE An organized collection of datafiles.

DATAFILE A file containing data, not programs.

DEBUGGING Trying to repair the damage caused by bad program design.

DEFAULT Value assumed when no explicit value is provided.

DIMENSION The range of an array subscript.

DISC Rigid or flexible rotating magnetic medium for long-term storage of programs and data.

EDITING The process of inserting, deleting and/or replacing lines of a program or file.

EOF End Of File.

ERROR MESSAGE Confusing description typed by the system when something goes wrong.

EXPRESSION Rule for computing a value from linked operators and operands.

FILE Program or data saved on disc or other backing-store.

FLOATING-POINT Notation expressing numbers as base times exponent, also called 'real'.

FLOWCHART Diagram showing the connections between steps in a procedure or program.

FUNCTION Procedure yielding a single result from one or more given values or arguments.

FUNCTION KEY A key assigned for a specific task.

FRONT-END Computer that handles I/O for another computer, or program that handles I/O for another program.

GARBAGE COLLECTION Procedure for reclaiming space taken by information that is no longer needed. Garbage collection normally takes place 'behind the scenes' when the system runs out of room, e.g. when creating temporary work-strings.

GLOBAL Global variables can be accessed from any part of a program: they are not local.

GLOSSARY List of mystifying explanations not containing the word you seek.

GRAPHICS The drawing of images on computer screens.

HACKER Person (usually male) who spends an inordinate amount of time hacking away at a computer – often with the intention of breaking into other people's remote computer systems via the telephone network.

HARDWARE The physical machinery of a computer system.

HEURISTIC A method of solving a problem which is quick but not foolproof.

HEXADECIMAL Relating to the number system to the base 16.

I/O Input/Output.

INFERENCE ENGINE That part of an expert system which draws conclusions from evidence given.

INPUT Getting information into a computer.

INSTRUCTION Sequence of symbols that tells the computer what to do next.

INTEGER Whole number.

INTERPRETER Program that interprets a program in Basic or another programming language which the computer cannot understand directly.

JARGON Words and phrases useful in advancing your chosen career.

KEYBOARD The mechanism for typing characters into the computer.

KEYWORD Word recognized as having a pre-defined meaning in a programming language.

KNOWLEDGE BASE The part of an expert system which stores its

	expertise, usually in the form of hundreds of decision rules.
LINE NUMBER	Number preceding a Basic statement used for sequencing and editing.
LOCAL	A local variable belongs to only one part of a program, for instance a function or procedure. Amstrad Basic does not have true local variables.
LOOP	Portion of a program that is executed repeatedly.
MACHINE CODE	The language that is directly understood by the computer's hardware without further translation.
MATRIX	An array with two dimensions.
MONITOR	Colour or monochrome display unit.
ON-LINE	Connected to a computer.
OPERAND	Value used by an operator in computing a result.
OPERATING SYSTEM	Software that controls the operation of a computer system while it is active.
OPERATOR	Keyword or special symbol that links operands into an expression which can be evaluated.
OUTPUT	Getting results out of the computer, typically in a form humans can understand.
PARAMETER	Value supplied to a procedure or instruction to tell it, for example, which of several options to choose. (See Argument.)
PERIPHERAL	Device attached to a computer and used for I/O, e.g. a printer.
PIXEL	The smallest addressable area of the display screen.
POINTER	Variable locating a record in a datafile or sometimes an element in another kind of data structure.
PROCEDURE	Collection of instructions grouped together to carry out a unitary task and given an identifying name. Routines are not generally considered to be procedures unless they have local variables and parameters. Hence Amstrad Basic does not support true procedures.
PROGRAM	Sequence of computer instructions.
PROGRAMMER	Harmless drudge.
RAM	Read-And-write Memory.
RANDOM ACCESS	Applied to files whose records may be processed in any order.
RECORD	Chunk of data in a file or table.
RECURSION	See Self-reference.
ROM	Read-Only-Memory.
ROUTINE	Subroutine, Function or Procedure.
RUNTIME	CPU time used by a program.

SECTOR A block of data on disc: AMSDOS uses sectors of 512 bytes in length.

SELF-REFERENCE See Self-reference.

SEQUENTIAL Sequential files may only be processed in serial order.

SERIAL See Sequential.

SOFTWARE Programs and routines that make the computer behave as an integrated system.

SORTING Arranging data in ascending or descending order.

STACK A data structure for 'stacking' information, where the last entry in is the first entry out.

STATEMENT Instruction line within a program.

STREAM A route for output from the computer.

STRING Sequence of characters.

SUBROUTINE Group of instructions for performing a specific purpose.

SUBSCRIPT Value which locates an element in an array.

SYSTEM Vague but prestigious word.

TIMESHARING Method of serving many users at once on one computer, by allocating short time-slices to each.

VALUE The information represented by a variable.

VARIABLE Name for a value that can change during program execution.

VDU Visual Display Unit.

VECTOR Array of one dimension.

WARM BOOT A warm boot re-initializes a computer system without shutting it down and therefore without necessarily losing the contents of memory. See Cold Boot.

WINDOW An independent section of the main screen area which behaves like a miniature screen in its own right.

WORKING MEMORY Place where a Basic program may be run, edited or listed.

Appendix H

Hints on further reading (Select bibliography)

The following admittedly incomplete list should help you get beyond the level of mere competence in Amstrad Basic. There are plenty of introductory Basic books for the Amstrad, but we have picked out mainly those which we think will expand your programming horizons. (Prices may be out of date by the time you read this.)

Atherton, Roy (1982), *Structured Programming with COMAL*, Ellis Horwood Ltd, Chichester (£9.50).

An excellent academically-oriented textbook on a language devised in the late 1970s to remedy Basic's major weaknesses by a Danish educationalist called Borge Christensen. Comal has never really taken off in the UK, but this book is still valuable for giving the Basic programmer a broader perspective.

Colwill, Steve (1985), *Games & Graphics Programming on the Amstrad Computers*, Micro Press (£9.95)

A well-written introduction to some of the more advanced techniques used in games and graphics programming.

Forsyth, Richard (1982), *Pascal at Work and Play*, Chapman and Hall, London (£7.50)

Sooner or later the Basic programmer feels the need to learn a 'grown-up' computer language. Pascal is one of the best choices, and this book is one of the better introductions to that language.

Forsyth, Richard and Morris, Brian (1985), *The BBC BASIC Idea*, Chapman and Hall/Methuen, London (£6.95).

The forerunner of the present work. Why not collect the set?

Forsyth, Richard and Naylor, Chris (1986), *The Hitch-hiker's Guide to Artificial Intelligence, Amstrad edition*, Chapman and Hall/Methuen, London (£8.75).

For those who want to take up some of the ideas on Artificial Intelligence introduced in the present book, this is one of the best-informed popular surveys of that subject. It also has plenty of interesting examples in Amstrad Basic to while away your time.

Gifford, Clive and Hartnell, Tim (1985), *The Amstrad Programmer's Guide*, Pitman, London (£6.95).

A cut above the usual rushed jobs that accompany the launch of any new microcomputer, this book is worth reading for a number of reasons, not least its explanations of sound and graphics.

Gray, Sean (1985), *Amstrad Book 2 – Sound, Graphics and Data Handling*, Glentop Publishers (£5.95).

Good value as a quick introduction to sound and graphics on the Amstrad, though less informative about what the author is pleased to call a 'database' system.

Lewis, T.G. (1979), *Software Engineering for Micros*, Hayden (£6.50).

A quirky but interesting romp round the field of structured programming techniques, concentrating mainly but not exclusively on low-level assembler-language problems.

Miller, Alan (1983), *Mastering CP/M*, Sybex (£13.50).

If you want to roam the plentiful pastures of CP/M, to which your CPC 664 or CPC 6128 provides access, this is a good book to read. It will take you beyond the basics which were all we had room for in the present slim volume.

Naylor, Chris (1983), *Build Your Own Expert System*, Sigma Technical Press (£6.95).

This was the first book to bring expert-systems methods to the masses. For this it provoked some displeasure in 'higher places': the trouble being that it is so clearly written, it makes the subject sound rather easy. Read it and enjoy it. All the examples are in a simple dialect of Basic which you should not find too hard to convert for the Amstrad.

Ogdin, Carol-Anne (1978), *Software Design for Microcomputers*, Reward/Prentice-Hall, Englewood Cliffs (£11.50).

A good solid treatment of the problems of designing well-structured software.

Zaks, Rodnay (1980), *Programming the Z80*, Sybex (£15.50).

For those who want to go on to machine-code with their Amstrads, this is as

good a guide as any to the language of the Z80 processor on which the Amstrad computers are based.

Zaks, Rodnay (1980), *Introduction to Pascal*, Sybex (£15.50).

The redoubtable Dr Zaks has not produced the cheapest book on Pascal, but he has produced one of the most thorough. It also deals with the major dialectic variant of Pascal, namely the UCSD version. The standard of programming textbooks has risen slowly over the last fifteen years, due in part to the efforts of Dr Zaks himself; but there is still plenty of rubbish about. If you want to see how it should be done properly, study this book: you may also stretch your mind beyond the confines of Basic in the process.

Answers to selected exercises

Model solutions are given here to all the odd-numbered exercises in the even-numbered chapters and to the even-numbered exercises in odd-numbered chapters. Program listings are accompanied, where appropriate, by sample printouts.

There is always more than one way of writing a program, so these answers, though correct, should not be regarded as definitive.

Exercise 3.2 Fibonacci series

```
10 REM ********************************
11 REM ** Exercise 3.2 :             **
12 REM ** FIBONACCI SERIES           **
15 REM ********************************
50 :
100 REM -- Fibonacci Numbers:
110 CLS
120 PRINT "---- FIBONACCI SERIES ----"
130 PRINT "Term No.","Term Value"
140 previous=0
150 last=1
160 toplimit=1000000
170 t%=0: term=0
180 WHILE term < toplimit
190   term = previous+last
200   t%=t%+1
210   PRINT t%,term
220   previous=last: last=term
230   WEND
250 PRINT "---- FIBONACCI SERIES ----"
300 END
320 :
```

```
---- FIBONACCI SERIES ----
Term No.      Term Value
 1             1
 2             2
 3             3
 4             5
 5             8
 6             13
```

```
 7              21
 8              34
 9              55
 10             89
 11             144
 12             233
 13             377
 14             610
 15             987
 16             1597
 17             2584
 18             4181
 19             6765
 20             10946
 21             17711
 22             28657
 23             46368
 24             75025
 25             121393
 26             196418
 27             317811
 28             514229
 29             832040
 30             1346269
---- FIBONACCI SERIES ----
```

Exercise 3.4 Array reversal

```
10 REM ******************************
11 REM ** Exercise 3.4 :            **
12 REM ** ARRAY REVERSAL            **
15 REM ******************************
50 :
100 REM -- Inverting an Array:
110 CLS
120 PRINT "---- ARRAY REVERSAL ----"
130 INPUT "How many items in the input ",size%
140 DIM datlist(size%)
150 FOR item%=1 TO size%
160   PRINT "Enter item no.";item%;
170   INPUT datlist(item%)
180   NEXT item%
190 PRINT
200 PRINT "---- OUTPUT OF DATA ----"
210 PRINT "Item No.","Original","Reverse"
220 FOR item%=size% TO 1 STEP -1
230   original%=size%+1-item%
240   PRINT original%,datlist(original%),datlist(item%)
250   NEXT item%
260 PRINT
300 END
320 :
---- ARRAY REVERSAL ----
How many items in the input 7
Enter item no. 1 ? 1
Enter item no. 2 ? 22
Enter item no. 3 ? 333
Enter item no. 4 ? 4444
Enter item no. 5 ? 55555
Enter item no. 6 ? 666666
Enter item no. 7 ? 7777777

---- OUTPUT OF DATA ----
Item No.      Original      Reverse
 1             1             7777777
 2             22            666666
 3             333           55555
 4             4444          4444
 5             55555         333
 6             666666        22
 7             7777777       1

Ready
```

Exercise 3.6 Asset depreciation

```
10 REM ********************************
11 REM ** Exercise 3.6 :              **
12 REM ** ASSET DEPRECIATION         **
15 REM ********************************
50 :
100 REM -- Depreciating an Asset:
110 CLS
120 PRINT "---- ASSET DEPRECIATION ----"
130 year%=0: loss=0
140 INPUT "Enter the asset's price: ",cost
150 INPUT "Enter its salvage value: ",salvage
160 INPUT "Enter its depreciation % ",rate
170 rate=rate/100: v=cost
175 PRINT
180 PRINT "Year     Depreciation     Value"
200 PRINT year%,loss,v
220 WHILE v>salvage
230   year%=year%+1
240   loss=loss + rate*v
250   v = cost-loss
260   IF v<salvage THEN v=salvage: loss=cost-salvage
270   PRINT year%,loss,v
280   WEND
300 END
320 :

---- ASSET DEPRECIATION ----
Enter the asset's price: 2500
Enter its salvage value: 1000
Enter its depreciation % 12.5

Year     Depreciation     Value
 0            0             2500
 1            312.5         2187.5
 2            585.9375      1914.0625
 3            825.195313    1674.80469
 4            1034.5459     1465.4541
 5            1217.72766    1282.27234
 6            1378.0117     1121.9883
 7            1500          1000
Ready
```

Exercise 4.1 Roots and powers

```
10 REM ********************************
11 REM ** Exercise 4.1 :              **
12 REM ** ROOTS & POWERS              **
15 REM ********************************
50 :
100 REM -- Roots and Powers:
110 CLS
120 PRINT "---- THE POWERS THAT BE ----"
150 GOSUB 1000 : REM define functions
160 INPUT "Number "; n
170 INPUT "Index  "; r
180 PRINT r;"th power of";n;"=";FNexpo(n,r)
190 PRINT r;"th root of ";n;"=";FNroot(n,r)
200 PRINT
300 END
330 :
1000 REM -- Subprogram with function definitions:
1010 DEF FNroot(a,b) = EXP(LOG(ABS(a))/b)*SGN(a)
1020 REM collapses with negative roots.
1030 :
1050 DEF FNexpo(a,b) = a^b
1060 RETURN
1070 :

---- THE POWERS THAT BE ----
Number ? 10
Index  ? -2
-2 th power of 10 = 0.01
-2 th root of  10 = 0.316227766

Ready

---- THE POWERS THAT BE ----
Number ? 4096
Index  ? 4
 4 th power of 4096 = 2.81475E+14
 4 th root of  4096 = 8

Ready
```

Exercise 4.3 Pascal's triangle

```
10 REM ********************************
11 REM ** Exercise 4.3 :              **
12 REM ** PASCAL'S TRIANGLE          **
15 REM ********************************
50 :
100 REM -- Pascal's Triangle:
110 CLS
120 PRINT "---- PASCAL'S TRIANGLE ----"
130 DIM triangle%(24)
140 triangle%(0)=0: triangle%(1)=1
150 w%=4
160 INPUT "Last Row Number "; rows%
170 GOSUB 1000 : REM generate triangle
200 PRINT
300 END
330 :
1000 REM -- Triangle routine:
1010 FOR r%=1 TO rows%
1020   GOSUB 2000 : REM make new row
1025   left%=w%*(rows%-r%) \ 2
1030   PRINT TAB(left%);
1040   FOR c%=1 TO r%
1050     PRINT TAB(left%+w%*c%);triangle%(c%);
1060     NEXT c%
1070   NEXT r%
1080 PRINT
1090 RETURN
1100 :
2000 REM -- Routine to make one line:
2010 FOR tp%=r% TO 2 STEP -1
2020   triangle%(tp%)=triangle%(tp%)+triangle%(tp%-1)
2030   NEXT tp%
2040 RETURN
2050 :

---- PASCAL'S TRIANGLE ----
Last Row Number ? 7
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5   10  10  5   1
    1   6   15  20  15  6   1

Ready
```

Exercise 4.5 Prime numbers

```
10 REM ********************************
11 REM ** Exercise 4.5 :              **
12 REM ** PRIME NUMBERS               **
15 REM ********************************
50 :
100 REM -- Prime Numbers:
110 CLS
120 PRINT "---- PRIME NUMBERS ----"
130 toplimit%=100: pcount%=1
140 DIM pnum%(toplimit%)
150 pnum%(0)=1: pnum%(1)=2
160 primenum%=3
170 PRINT "The first";toplimit%;"primes are:"
200 WHILE pcount%<toplimit%
210   divisors%=0
220   GOSUB 1000 : REM test latest number
230   IF divisors%=0 THEN PRINT primenum%;: pcount%=pcount%+1
240   primenum%=primenum%+2
250   WEND
260 REM steps through odd numbers.
270 PRINT
300 END
330 :
1000 REM -- Routine to test primenum%:
1010 pmax%=SQR(primenum%)
1020 FOR p%=1 TO pcount%
1030   IF (primenum% MOD pnum%(p%) = 0) THEN divisors%=1: p%=pco
unt%
1040   IF pnum%(p%)>=pmax% THEN p%=pcount%
1050   NEXT p%
1060 REM loop ends early if p%>pmax% or divisor found.
1070 pnum%(pcount%+1)=primenum%
1075 RETURN
1080 :

---- PRIME NUMBERS ----
The first 100 primes are:
 3   5   7   11   13   17   19   23   29   31
 37   41   43   47   53   59   61   67   71   73
 79   83   89   97   101   103   107   109
 113   127   131   137   139   149   151   157
 163   167   173   179   181   191   193   197
 199   211   223   227   229   233   239   241
 251   257   263   269   271   277   281   283
 293   307   311   313   317   331   337   347
 349   353   359   367   373   379   383   389
 397   401   409   419   421   431   433   439
 443   449   457   461   463   467   479   487
 491   499   503   509   521   523   541
Ready
```

Exercise 4.7 Perfect numbers

```
10 REM ********************************
11 REM ** Exercise 4.7 :              **
12 REM ** PERFECT NUMBERS             **
15 REM ********************************
50 :
100 REM -- Perfect Numbers:
110 CLS
120 PRINT "---- PERFECT NUMBERS ----"
130 toplimit%=100
140 DIM fact%(toplimit%)
150 possible%=3
160 perfects%=0
200 WHILE perfects%<2
210   possible%=possible%+1
215   PRINT "TESTING ";possible%;
220   pointer%=1: tote%=1
230   GOSUB 1000 : REM test for perfection
240   IF perfect% THEN PRINT " = PERFECT": perfects%=perfects%+1
 ELSE PRINT " = IMPERFECT"
250   WEND
270 PRINT
300 END
330 :
1000 REM -- Routine to test perfection:
1010 sqrt%=SQR(possible%)
1020 FOR numb%=2 TO sqrt%
1030   GOSUB 2000 : REM get factors
1040   IF tote%>possible% THEN numb%=sqrt%: REM quit loop
1050   NEXT numb%
1060 IF tote%=possible% THEN perfect%=1 ELSE perfect%=0
1070 RETURN
1080 :
2000 REM -- Factorization routine:
2010 IF possible% MOD numb% <> 0 THEN RETURN
2020 fact%(pointer%)=numb%
2030 fact%(pointer%+1)=(possible% \ numb%)
2040 tote%=tote%+fact%(pointer%)+fact%(pointer%+1)
2050 pointer%=pointer%+2
2070 RETURN
2075 REM factors stored in fact%().
2080 :
```

```
---- PERFECT NUMBERS ----
TESTING  4  = IMPERFECT
TESTING  5  = IMPERFECT
TESTING  6  = PERFECT
TESTING  7  = IMPERFECT
TESTING  8  = IMPERFECT
TESTING  9  = IMPERFECT
TESTING  10  = IMPERFECT
TESTING  11  = IMPERFECT
TESTING  12  = IMPERFECT
TESTING  13  = IMPERFECT
TESTING  14  = IMPERFECT
TESTING  15  = IMPERFECT
TESTING  16  = IMPERFECT
TESTING  17  = IMPERFECT
TESTING  18  = IMPERFECT
TESTING  19  = IMPERFECT
```

```
TESTING  20  = IMPERFECT
TESTING  21  = IMPERFECT
TESTING  22  = IMPERFECT
TESTING  23  = IMPERFECT
TESTING  24  = IMPERFECT
TESTING  25  = IMPERFECT
TESTING  26  = IMPERFECT
TESTING  27  = IMPERFECT
TESTING  28  = PERFECT
```

Exercise 5.2 Procrustes

```
10 REM ******************************
11 REM ** Exercise 5.2 :            **
12 REM ** PROCRUSTES                **
15 REM ******************************
50 :
100 REM -- Procrustean Program:
110 CLS
120 PRINT "---- BED OF PROCRUSTES ----"
140 padchar$=".": text$=padchar$
144 REM change padchar$ for different padding (e.g. space).
150 WHILE text$<>""
160   INPUT "Enter your string ",text$
170   INPUT "Enter desired length ", size%
180   INPUT "Enter 'L' for leading spaces ",padding$
190   IF UPPER$(padding$)="L" THEN lpad%=1 ELSE lpad%=0
200   PRINT "!!";text$;"!!"
210   PRINT "  BECOMES"
220   GOSUB 1000 : REM chop and pad
240   PRINT "!!";newtext$;"!!"
250   WEND
270 PRINT
300 END
330 :
1000 REM -- Chop & pad routine:
1010 REM given text$ produces newtext$
1020 s%=LEN(text$)
1030 IF s%>size% THEN newtext$=LEFT$(text$,size%): RETURN
1040 gaps$=STRING$(size%-s%,padchar$)
1050 IF lpad% THEN newtext$=gaps$+text$ ELSE newtext$=text$+gaps
$
1070 RETURN
1080 :
```

Exercise 5.4 Left and right string

```
10 REM ********************************
11 REM ** Exercise 5.4 :             **
12 REM ** LEFT AND RIGHT STRINGS     **
15 REM ********************************
50 :
100 REM -- Simulated LEFT$ and RIGHT$:
110 CLS
120 PRINT "---- HEADS AND TAILS ----"
140 text$="*"
148 REM -- Main loop:
150 WHILE text$<>""
160   INPUT "Enter your string ",text$
170   INPUT "Enter substring length ", size%
180   PRINT "Leftmost";size%;"characters are : ";
190   GOSUB 1000 : REM left string
200   PRINT newtext$
210   PRINT "Rightmost";size%;"characters are: ";
220   GOSUB 1200 : REM right string
230   PRINT newtext$
240   PRINT
250   WEND
270 PRINT
300 END
330 :
1000 REM -- Left-string routine:
1010 newtext$=MID$(text$,1,size%)
1020 RETURN
1050 :
1200 REM -- right-string routine:
1202 IF size%>LEN(text$) THEN newtext$=text$: RETURN
1210 newtext$=MID$(text$,LEN(text$)-size%+1)
1220 RETURN
1250 :

---- HEADS AND TAILS ----
Enter your string Julius Caesar
Enter substring length 7
Leftmost 7 characters are : Julius
Rightmost 7 characters are:  Caesar

Enter your string Amstrad stinks!
Enter substring length 6
Leftmost 6 characters are : Amstra
Rightmost 6 characters are: tinks!

Enter your string Bonk
Enter substring length 10
Leftmost 10 characters are : Bonk
Rightmost 10 characters are: Bonk

Enter your string
```

Exercise 5.6 Instring

```
10 REM ********************************
11 REM ** Exercise 5.6 :              **
12 REM ** INSTRING                    **
15 REM ********************************
50 :
100 REM -- Simulated INSTR with wildcards:
110 CLS
120 PRINT "---- WILD-CARD INSTRING ----"
130 wild$="?"
140 text$="*"
148 REM -- Main loop:
150 WHILE text$<>""
160   INPUT "Enter the string ",text$
170   INPUT "Enter a subtring ",seek$
180   GOSUB 1000 : REM look for it
190   IF instring%=0 THEN PRINT seek$;" does not occur in ";text
$
200   IF instring%>0 THEN PRINT seek$;" occurs at position";inst
ring%;"of ";text$
210   PRINT
220   WEND
270 PRINT
300 END
330 :
1000 REM -- Instring subroutine:
1010 len1%=LEN(text$)
1020 len2%=LEN(seek$)
1025 instring%=0
1030 IF len1%*len2%=0 THEN RETURN
1040 FOR ps%=1 TO len1%-len2%+1
1050   ch$=LOWER$(MID$(text$,ps%,len2%))
1055   mismatch%=0
1060   FOR px%=1 TO len2%
1070     cx$=MID$(ch$,px%,1)
1080     cy$=LOWER$(MID$(seek$,px%,1))
1090     IF cx$<>cy$ AND cy$<>wild$ THEN mismatch%=1: px%=len2%
: REM quit inner loop
1100     NEXT px%
1110   IF mismatch%=0 THEN instring%=ps%: ps%=len1% : REM quit l
oop
1120   NEXT ps%
1140 RETURN
1150 :

---- WILD-CARD INSTRING ----
Enter the string Wild Cardinals!
Enter a subtring wild card
wild card occurs at position 1 of
Wild Cardinals!

Enter the string Hello Folks!
Enter a subtring ?lo??ol?
?lo??ol? occurs at position 3 of
Hello Folks!

Enter the string Get lost
Enter a subtring get stuffed
get stuffed does not occur in Get lost
```

```
Enter the string Goodbye
Enter a subtring bye bye!
bye bye! does not occur in Goodbye

Enter the string
Enter a subtring
 does not occur in


Ready
```

Exercise 6.1 Reaction timer

```
10 REM ********************************
11 REM ** Exercise 6.1 :              **
12 REM ** REACTION TIMER              **
15 REM ********************************
50 :
100 REM -- Reaction Timer:
110 CLS
120 PRINT "---- TYPING TUTORIAL ----"
130 PRINT "(1) The screen will clear."
140 PRINT "(2) Text will suddenly appear."
150 PRINT "(3) You must copy the text."
160 PRINT "(4) Your reaction will be timed."
170 PRINT
175 PRINT "Press a key to begin."
180 text$="The Slow Blue Fox"
188 a$=INKEY$: IF a$="" THEN GOTO 188
190 REM -- Main Loop:
199 a$="$"
200 WHILE a$<>"!"
210   CLS
220   GOSUB 1000 : REM display text$
230   PRINT
240   PRINT USING "Initial reaction time = ####.##";reactime
250   PRINT USING "Typing time (seconds) = ####.##";typetime
260   PRINT USING "Accuracy (%)          = ####.##";percent
270   PRINT: PRINT "Hit ! to stop, other keys to go on."
280   a$=INKEY$: IF a$="" THEN GOTO 280
290   WEND
300 END
330 :
1000 REM -- Display routine:
1010 r%=INT(RND*15)+1: c%=INT(8*RND)+2
1020 FOR d%=1 TO 200+RND*800
1030   NEXT d% : REM delay
1040 LOCATE c%,r%: PRINT text$
1050 t0=TIME
1060 ch$=INKEY$: IF ch$="" THEN GOTO 1060
1070 t1=TIME: PRINT ch$;
1080 LINE INPUT "", in$
1090 t2=TIME
1100 in$=ch$+in$
1110 GOSUB 1600 : REM scoring
1120 reactime=(t1-t0)/300
1130 typetime=(t2-t1)/300
1150 RETURN
```

```
1160 :
1600 REM -- Scoring routine:
1610 size%=LEN(text$)
1620 s%=0
1640 FOR pp%=1 TO size%
1650   IF MID$(text$,pp%,1)=MID$(in$,pp%,1) THEN s%=s%+1
1670   NEXT pp%
1680 percent=s%*100/size%
1700 RETURN
1720 :
```

```
        The Slow Blue Fox
thngbe slow blue dog!

Initial reaction time =     0.86
Typing time (seconds) =     4.61
Accuracy (%)          =     5.88

Hit ! to stop, other keys to go on.

        The Slow Blue Fox
The Slow Blue Fox

Initial reaction time =     0.64
Typing time (seconds) =     3.34
Accuracy (%)          =   100.00

Hit ! to stop, other keys to go on.
Ready
```

Exercise 6.3 Compound interest

```
10 REM ********************************
11 REM ** Exercise 6.3 :              **
12 REM ** COMPOUND INTEREST          **
15 REM ********************************
50 :
100 REM -- Interest Tables:
101 ouch%=8
110 CLS
120 PRINT "---- INTEREST/INFLATION ----"
125 WIDTH 80
130 PRINT
140 DIM a(12)
150 PRINT "What is initial amount (1 to 5000)";
160 INPUT amount1
170 IF amount1<1 OR amount1>5000 THEN PRINT "Try again": GOTO 15
0
175 lineform$="##  ####.##  ####.##  ####.##  ####.##  ####.##  ####.##
####.##"
180 REM -- Set initial values:
190 FOR i%=2 TO 12 STEP 2
200   a(i%)=amount1: NEXT i%
210 REM -- Print headings:
220 GOSUB 1000
230 REM -- Main Loop:
250 FOR y%=1 TO 25
260   FOR i%=2 TO 12 STEP 2
270     LET interest=a(i%)*i%/100
280     LET a(i%)=a(i%)+interest
290     NEXT i%
300   PRINT #ouch%, USING lineform$; y%,a(2),a(4),a(6),a(8),a(10
),a(12)
310   NEXT y%
320 PRINT#ouch%
330 END
360 :
1000 REM -- Header routine:
1010 PRINT#ouch%
1020 PRINT#ouch%, "COMPOUND INTEREST TABLE:"
1030 PRINT #ouch%, "(Initial Amount =";amount1;")"
1040 PRINT#ouch%
1050 PRINT#ouch%,  "Period";
1060 FOR i%=2 TO 12 STEP 2
1070   PRINT #ouch%, TAB(4.5*i%-1);i%;"%";
1080   NEXT i%
1090 PRINT#ouch%
1100 RETURN
1110 :
```

```
COMPOUND INTEREST TABLE:
(Initial Amount = 1000 )

Period   2 %       4 %       6 %       8 %       10 %      12 %
 1   1020.00   1040.00   1060.00   1080.00   1100.00   1120.00
 2   1040.40   1081.60   1123.60   1166.40   1210.00   1254.40
 3   1061.21   1124.86   1191.02   1259.71   1331.00   1404.93
 4   1082.43   1169.86   1262.48   1360.49   1464.10   1573.52
 5   1104.08   1216.65   1338.23   1469.33   1610.51   1762.34
 6   1126.16   1265.32   1418.52   1586.87   1771.56   1973.82
 7   1148.69   1315.93   1503.63   1713.82   1948.72   2210.68
```

8	1171.66	1368.57	1593.85	1850.93	2143.59	2475.96
9	1195.09	1423.31	1689.48	1999.00	2357.95	2773.08
10	1218.99	1480.24	1790.85	2158.93	2593.74	3105.85
11	1243.37	1539.45	1898.30	2331.64	2853.12	3478.55
12	1268.24	1601.03	2012.20	2518.17	3138.43	3895.98
13	1293.61	1665.07	2132.93	2719.62	3452.27	4363.49
14	1319.48	1731.68	2260.90	2937.19	3797.50	4887.11
15	1345.87	1800.94	2396.56	3172.17	4177.25	5473.57
16	1372.79	1872.98	2540.35	3425.94	4594.97	6130.39
17	1400.24	1947.90	2692.77	3700.02	5054.47	6866.04
18	1428.25	2025.82	2854.34	3996.02	5559.92	7689.97
19	1456.81	2106.85	3025.60	4315.70	6115.91	8612.76
20	1485.95	2191.12	3207.14	4660.96	6727.50	9646.29
21	1515.67	2278.77	3399.56	5033.83	7400.25	%10803.85
22	1545.98	2369.92	3603.54	5436.54	8140.27	%12100.31
23	1576.90	2464.72	3819.75	5871.46	8954.30	%13552.35
24	1608.44	2563.30	4048.93	6341.18	9849.73	%15178.63
25	1640.61	2665.84	4291.87	6848.48	%10834.71	%17000.06

Exercise 6.5 Histogram program

```
10 REM ********************************
11 REM ** Exercise 6.5 :              **
12 REM ** HISTOGRAM PROGRAM           **
15 REM ********************************
50 :
100 REM -- Sideways Histogram:
101 ouch%=8 : REM printer channel
110 CLS
120 PRINT#ouch%, "---- HISTOGRAM ----"
125 WIDTH 80
130 PRINT
140 DIM x(100)
150 star$="*": RESTORE
160 READ nscores%
170 IF nscores%<1 OR nscores%>100 THEN PRINT "Try again": STOP
180 REM -- Get and Scale values:
190 GOSUB 1000
200 REM -- Heading:
210 PRINT #ouch%, "Scaled from 0 to";xmax;"units:"
220 PRINT #ouch%, "One star =";xmax/wmax;"units."
230 PRINT #ouch%
250 FOR n%=1 TO nscores%
260   PRINT #ouch%, n%;TAB(6);"|";
270   tabwidth%=x(n%)*wmax/xmax
280   FOR star%=1 TO tabwidth%
290     PRINT #ouch%, star$;
300     NEXT star%
310   PRINT#ouch%, ROUND(x(n%))
320   NEXT n%
330 PRINT
350 END
360 :
1000 REM -- Data input scaling routine:
1010 wmax=64
1020 xmax=0
1030 FOR n%=1 TO nscores%
1040   READ x(n%)
1050   IF x(n%)<0 OR x(n%)>999999 THEN PRINT "Data out of range!":
       STOP
1060   IF x(n%)>xmax THEN xmax=x(n%)
1070   NEXT n%
1080 RETURN
1100 :
1200 REM -- Test data:
1210 DATA 12
1220 DATA 115.55
1230 DATA 1123.55
1240 DATA 1066.26
1250 DATA 1439.13
1260 DATA 3567.87
1270 DATA 3069.75
1280 DATA 1250.97
1290 DATA 493.79
1300 DATA 1418.72
1310 DATA 2048.70
1320 DATA 935.25
1330 DATA 137
```

```
---- HISTOGRAM ----
Scaled from 0 to 3567.87 units:
One star = 55.7479688 units.

 1   |** 116
 2   |******************** 1124
 3   |******************* 1066
 4   |************************* 1439
 5   |************************************************************
     |**** 3568
 6   |********************************************************* 3070
 7   |********************** 1251
 8   |********* 494
 9   |************************ 1419
 10  |************************************ 2049
 11  |***************** 935
 12  |** 137
```

Exercise 7.2 Word counter

```
10 REM ******************************
11 REM ** Exercise 7.2 :            **
12 REM ** WORD COUNTER              **
15 REM ******************************
50 :
100 REM -- Program to Count Words:
110 CLS
120 PRINT "---- WORD COUNTER ----"
130 word%=0 : REM false
140 sp$=" ": dash$="-"
150 INPUT "Text File Name ",f$
160 OPENIN f$
170 wc%=0: cc%=0 : REM counters
175 lc%=0
180 REM -- Main Loop:
200 WHILE NOT EOF
210   LINE INPUT #9, text$
220   last%=LEN(text$)
230   FOR c%=1 TO last%
240     ch$=UPPER$(MID$(text$,c%,1))
250     cc%=cc%+1
260     alphanum%=0
270     IF ch$>="A" AND ch$<="Z" THEN alphanum%=1
280     IF ch$>="0" AND ch$<="9" OR ch$=dash$ THEN alphanum%=2
290     IF word%=0 AND alphanum%=1 THEN word%=1
300     IF word% AND NOT alphanum% THEN word%=0: wc%=wc%+1
320     NEXT c%
322   lc%=lc%+1
325   WEND
330 PRINT
350 CLOSEIN
360 PRINT f$;" has ";cc%;" characters."
370 PRINT f$;" has ";wc%;" words."
375 PRINT f$;" has ";lc%;" lines."
380 PRINT
400 END
440 :


---- WORD COUNTER ----
Text File Name text

text has  231  characters.
text has  16  words.
text has  8  lines.

Ready
```

Exercise 7.4 Menu program

```
10 REM ********************************
11 REM ** Exercise 7.4 :              **
12 REM ** MENU PROGRAM                **
15 REM ********************************
50 :
100 REM -- Menu Program:
110 MODE 2
120 PRINT "---- MAIN MENU ----"
125 PRINT
140 READ choices%
150 DIM progname$(choices%)
160 FOR counter%=1 TO choices%
170   READ progname$(counter%),text$
180   PRINT CHR$(64+counter%),progname$(counter%),text$
200   NEXT counter%
220 PRINT: PRINT "---- HIT THE LETTER OF YOUR CHOICE ----"
230 ch$="*"
250 WHILE ch$<"A" OR ch$>CHR$(64+choices%)
270   ch$=INKEY$: IF ch$="" THEN GOTO 270
275   ch$=UPPER$(ch$)
280   WEND
290 ch%=ASC(ch$)-64
300 PRINT: PRINT ch$;" ---> ";progname$(ch%)
310 CHAIN progname$(ch%)
400 END
440 :
1000 REM -- Menu Data:
1010 DATA 4
1020 DATA LIST,Lists a data file
1030 DATA SORT,Sorts a data file
1040 DATA KILL,Deletes a data file
1050 DATA ZONK,Does something vile
1100 :
```

```
---- MAIN MENU ----

A             LIST          Lists a data file
B             SORT          Sorts a data file
C             KILL          Deletes a data file
D             ZONK          Does something vile

---- HIT THE LETTER OF YOUR CHOICE ----

C ---> KILL
```

Exercise 7.6 Revised golf-handicapper

```
10 REM *********************************
11 REM ** Exercise  7.6 :             **
12 REM ** REVISED GOLF-HANDICAPPER    **
15 REM *********************************
20 REM -- Must RUN "BANKMAN" first.
50 ON ERROR GOTO 500 : REM error trap
60 :
100 REM -- GOLFERS:
101 ouch%=0 : REM 8=printer, 0=screen.
110 MODE 1 : BORDER 15
120 recsize%=24
125 gaps$=SPACE$(recsize%)
130 chan%=9
140 INPUT "File Name "; f$
150 OPENIN f$
160 GOSUB 1000 : REM load data from file
170 PRINT: PRINT "Data from file : ";f$
180 command%=99
190 REM -- Main Loop:
200 WHILE command% <> 0
210   GOSUB 1200 : REM menu
220   IF command%=1 THEN GOSUB 1500 : REM new player
230   IF command%=2 THEN GOSUB 1700 : REM old player
240   IF command%=3 THEN GOSUB 2000 : REM show player
250   IF command%=4 THEN GOSUB 2300 : REM handicap
260   GOSUB 2200 : REM pause
270   WEND
280 CLOSEIN
300 GOSUB 3000 : REM write data back out
320 PRINT: PRINT "Bye!"
330 END
333 :
500 REM -- error trap section:
510 IF ERR=32 AND ERL=150 THEN GOSUB 600 ELSE ON ERROR GOTO 0
530 RESUME 150 : REM resume
550 :
600 REM -- create new file (1st time only):
605 n$=gaps$
610 OPENOUT f$
615 WRITE#chan%, n$,-99
620 CLOSEOUT
630 RETURN
650 REM contains one dummy record.
660 :
1000 REM -- Data input routine:
1010 recs%=0: r%=0
1020 |BANKOPEN,recsize%
1040 WHILE NOT EOF
1050   INPUT#chan%, name$,h
1060   h$=LEFT$(STR$(h)+gaps$,recsize%)
1070   name$=LEFT$(name$+gaps$,recsize%)
1080   |BANKWRITE,@r%,name$
1090   IF r%=-1 THEN PRINT "UGH!"
1100   |BANKWRITE,@r%,h$
1110   IF r%=-1 OR r%=-2 THEN PRINT "EH!?": STOP
1120   recs%=recs%+2
1130   WEND
1140 CLOSEIN
1150 PRINT recs%/2;"items read from file ";f$
```

```
1155 PRINT "Press any key to go on :"
1156 c$=INKEY$: IF c$="" THEN GOTO 1156
1160 CLEAR INPUT: RETURN
1170 :
1200 REM -- Menu subroutine:
1210 CLS: PRINT
1220 PRINT TAB(16);"Options:"
1230 PRINT TAB(16);"========"
1240 PRINT TAB(16);"[";f$;"]"
1250 PRINT
1260 PRINT "1 .... Add a new player"
1270 PRINT "2 .... Delete an old player"
1280 PRINT "3 .... Show a player's details"
1290 PRINT "4 .... Revise a player's handicap"
1300 PRINT: PRINT "No. of option (0 to quit) : ";
1310 c$=INKEY$: IF c$="" THEN GOTO 1310
1313 command%=ASC(c$)-48
1320 PRINT c$
1330 RETURN
1340 :
1500 REM -- New-player routine:
1510 PRINT: PRINT "Adding new player:"
1520 at%=-1 : p%=0
1530 WHILE at%<0 AND p%<recs%
1540   |BANKREAD,@r%,name$,p%
1550   |BANKREAD,@r%,h$,p%+1
1560   IF r%<0 THEN PRINT "Gasp": STOP
1570   IF VAL(h$)<0 THEN at%=p%
1580   p%=p%+2
1590   WEND
1595 IF at%<0 THEN at%=recs%: recs%=recs%+2
1600 REM -- Insert at at%:
1610 h=-99 : name$=""
1620 WHILE (h<0 OR h>36) OR name$=""
1630   LINE INPUT "Player's name: ", name$
1640   LINE INPUT "Handicap is  : ", h$
1650   name$=LEFT$(name$+gaps$,recsize%)
1660   h=VAL(h$)
1670   WEND
1680 GOSUB 4000 : REM write a record
1695 PRINT "inserted at location ";at%
1696 RETURN
1699 :
1700 REM -- Record-deletion routine:
1710 PRINT: PRINT "Record deletion:"
1720 GOSUB 5000
1730 GOSUB 4400 : REM show it
1740 PRINT "OK to delete it (Y=Yes) ";
1750 c$=INKEY$: IF c$="" THEN GOTO 1750
1760 ok%=(c$="Y") OR (c$="y")
1765 PRINT c$
1770 IF NOT ok% THEN PRINT "Not erased.": RETURN
1780 REM -- mark as dead with -ve handicap:
1788 PRINT
1790 h=-99
1800 GOSUB 4000 : REM write record
1810 PRINT "Player ";name$;" erased."
1820 GOSUB 2200 : REM delay
1850 RETURN
1860 :
2000 REM -- Show-details routine:
2010 PRINT: PRINT "Show details for player:"
```

```
2020 GOSUB 5000 : REM -- get record number
2030 IF at%<0 THEN PRINT "Sorry!": RETURN
2040 IF at%>recs% THEN PRINT "Sorry!": RETURN
2050 GOSUB 4400 : REM display routine
2060 GOSUB 2200 : REM wait
2070 RETURN
2080 :
2200 REM -- Delay routine:
2202 REM uses w%, w$
2210 w%=0 : w$=""
2220 WHILE w%<222 AND w$=""
2230   w$=INKEY$
2240   w%=w%+1
2250   WEND
2260 RETURN
2270 :
2300 REM -- New-handicap routine:
2310 PRINT: PRINT "Revision of handicap data:"
2320 GOSUB 5000 : REM get record
2330 GOSUB 4400 : REM display it
2340 PRINT "OK to alter handicap ? ";
2350 c$=INKEY$: IF c$="" THEN GOTO 2350
2360 ok%=(c$="Y") OR (c$="y")
2370 IF NOT ok% THEN RETURN
2380 IF h<0 THEN RETURN : REM dud record
2390 PRINT
2400 INPUT "Latest score was : ", s
2410 INPUT "Par for the course:", p%
2420 s = INT(s-p%) : REM difference from par
2430 GOSUB 2500 : REM compute new handicap
2440 PRINT: PRINT "New handicap for ";name$
2450 PRINT h;" (playing off ";INT(h+0.5);")"
2460 GOSUB 4000 : REM write it.
2470 PRINT "Press any key to go on:"
2475 c$=INKEY$: IF c$="" THEN 2475
2480 RETURN
2490 :
2500 REM -- Handicap-revision routine:
2510 d = s - INT(h+0.5)
2520 IF d=0 THEN RETURN
2530 IF d>0 AND h<=5 THEN h=h+0.1: RETURN
2540 IF d>0 AND h>5 THEN h=h+0.2: RETURN
2550 REM -- gets here if better than usual:
2560 IF h<=0 THEN h=0: RETURN
2570 WHILE d < 0
2580   x = 0.1
2590   IF h > 20.4 THEN x=x+0.1
2600   IF h > 12.4 THEN x=x+0.1
2610   IF h > 5.4 THEN x=x+0.1
2620   h = h - x
2630   d = d + 1
2640   WEND
2650 RETURN
2660 REM with h as new handicap.
2670 :
3000 REM -- Write-file routine:
3010 p%=0
3020 OPENOUT f$
3030 WHILE p%<recs%
3040   |BANKREAD,@r%,name$,p%
3050   |BANKREAD,@r%,h$,p%+1
3060   IF r%<0 THEN STOP
```

```
3070   h=VAL(h$)
3080   WRITE#chan%, name$,h
3090   p%=p%+2
3100   WEND
3110 CLOSEOUT
3120 PRINT recs%/2;" items dumped to ";f$
3130 PRINT
3140 RETURN
3150 :
4000 REM -- Write-record routine:
4010 REM puts name$,h at at%
4020 h$=LEFT$(STR$(h)+gaps$,recsize%)
4030 |BANKWRITE,@r%,name$,at%
4040 |BANKWRITE,@r%,h$,at%+1
4050 IF r%<0 THEN STOP
4060 RETURN
4070 :
4200 REM -- Read-record routine:
4210 REM gets name$,h from at%
4215 name$=gaps$
4220 |BANKREAD,@r%,name$,at%
4230 |BANKREAD,@r%,h$,at%+1
4240 IF r%<0 THEN STOP
4250 h=VAL(h$)
4260 RETURN
4270 :
4400 REM -- record-display routine:
4410 REM shows record at at%:
4420 GOSUB 4200
4430 IF h<0 THEN PRINT "EMPTY RECORD!": RETURN
4440 PRINT
4450 PRINT "Player name : ";name$
4460 PRINT "Handicap is : ";h
4470 PRINT "Playing off : ";INT(h+0.5)
4480 PRINT
4490 RETURN
4500 :
5000 REM -- Routine to get record-number:
5010 PRINT: ok%=0
5020 WHILE ok%=0
5030   LINE INPUT "Player's name "; n$
5040   n$=LEFT$(n$,12)
5050   IF n$="" THEN GOTO 5030
5060   |BANKFIND,@r%,n$,0,recs%
5066   n%=r%
5070   IF n%>=0 THEN ok%=1
5080   WEND
5090 at%=n%: RETURN
5100 :
```

Exercise 7.8 Indexer program

```
10 REM ******************************
11 REM ** Listing 7.8 :            **
12 REM ** INDEXER PROGRAM          **
15 REM ******************************
60 :
100 REM -- Shell-sort for book index:
101 ouch%=8 : REM 8=printer, 0=screen.
110 MODE 1 : BORDER 15
120 READ n%
130 DIM term$(n%), page$(n%)
133 DIM p%(n%)
140 FOR i%=1 TO n%
150   READ term$(i%),page$(i%)
180   p%(i%)=i%
190   PRINT i%,term$(i%)
200   NEXT i%
210 REM -- Data now read in.
220 GOSUB 400 : REM sort them out
230 PRINT #ouch%
233 PRINT #ouch%,"Appendix I -- Index of Example Programs:"
240 PRINT #ouch%, "INDEX OF";n%;"ITEMS:"
244 PRINT #ouch%
250 FOR i%=1 TO n%
260   PRINT #ouch%, i%;TAB(6);term$(i%);TAB(33);page$(p%(i%))
270   NEXT i%
280 PRINT #ouch%
290 PRINT#ouch%, "La.b is Listing b in Chapter a;"
295 PRINT#ouch%, "Xc.d is Exercise d in Chapter c."
300 END
320 REM -- for output see Appendix I.
330 :
400 REM -- Shell-sort subroutine:
410 m% = n%
420 WHILE m% > 1
430   m% = INT(m% / 2)
440   IF m% MOD 2 = 0 THEN m%=m%-1
450   FOR i%=m%+1 TO n%
460     FOR j%=i% TO m%+1 STEP -m%
470       IF UPPER$(term$(j%))<UPPER$(term$(j%-m%)) THEN GOSUB 800
 ELSE j%=0
480       NEXT
490     NEXT i%
500   WEND
520 RETURN
550 :
800 REM -- swap routine:
810 r%=j%-m%
820 t%=p%(j%): p%(j%)=p%(r%): p%(r%)=t%
830 temp$=term$(j%)
840 term$(j%)=term$(r%)
850 term$(r%)=temp$
860 RETURN
870 REM uses t,temp$,r%,j%
880 :
1000 REM -- Data for sorting:
1010 DATA 55
1020 DATA Probability Calculations,L1.1
1030 DATA Easter Day Calculations,L2.1
1040 DATA Centigrade to Fahrenheit,L3.1
```

```
1050 DATA Temperature Converter,L3.2
1060 DATA Russian Roulette,L3.3
1070 DATA Simple Selection Sort,L3.4
1080 DATA Reversing Numbers,L4.1
1090 DATA The Ladder of Recursion,L4.2
1100 DATA Rounding a Numeric Value,L4.3
1110 DATA Bisector Program,L4.4
1120 DATA String Input,L5.1
1130 DATA ASCII Values,L5.2
1140 DATA The ASCII Character Set,L5.3
1150 DATA Comparing Strings,L5.4
1160 DATA String Functions,L5.5
1170 DATA Welcome Back,L5.6
1180 DATA Number Naming, L5.7
1190 DATA Input Demonstration,L6.1
1200 DATA Input Validation,L6.2
1210 DATA Party Invitations,L6.3
1220 DATA PRINT USING with Numbers,L6.4
1230 DATA Screen-Based Form-Filling,L6.5
1240 DATA Very Basic Expert System,L6.6
1250 DATA Sales Datafile,L7.1
1260 DATA Sequential File Creation,L7.2
1270 DATA Sequential File Listing,L7.3
1280 DATA Golf-Club Handicapper,L7.4
1290 DATA Shell Sort,L7.5
1300 DATA Starburst,L8.1
1310 DATA Coloured Balloons,L8.2
1320 DATA Pentatonic Player,L8.3
1330 DATA Rat-Maze Program,L8.4
1340 DATA Binary Tree Creation,L9.1
1350 DATA Recursiive Patterns,L9.2
1360 DATA Code-Game -- Version 1,L10.2
1370 DATA Code-Game -- Version 2,L10.2
1380 DATA Fibonacci Series,X3.2
1390 DATA Array Reversal,X3.4
1400 DATA Asset Depreciation,X3.6
1410 DATA Roots and Powers,X4.1
1420 DATA Pascal´s Triangle,X4.3
1430 DATA Prime Numbers,X4.5
1440 DATA Perfect Numbers,X4.7
1450 DATA Procrustes,X5.2
1460 DATA Left & Right Strings,X5.4
1470 DATA Instring,X5.6
1480 DATA Reaction Timer,X6.1
1490 DATA Compound Interest,X6.3
1500 DATA Histogram Program,X6.5
1510 DATA Word Counter,X7.2
1520 DATA Menu Program,X7.4
1530 DATA Revised Golf-Handicapper,X7.6
1540 DATA Polygon Plot,X8.1
1550 DATA Text-Screen Dump,X8.3
1560 DATA Indexer Program,X7.8
```

Exercise 8.1 Polygon plot

```
10 REM ********************************
11 REM ** Exercise  8.1 :             **
12 REM ** POLYGON PLOT                **
15 REM ********************************
60 :
100 REM -- Polygon Plotter:
101 RAD
110 MODE 1 : BORDER 15
120 h=200: v=128
130 size=64: edge=10
140 flag%=2
150 GOSUB 1000 : REM polyplot routine
200 END
220 :
1000 REM -- Polygon plotting routine:
1010 REM uses:
1020 ph=PI/edge
1030 r=SIN(ph)*size
1040 MOVE h,v-size
1050 MOVER r*COS(ph),r*SIN(ph)
1060 FOR e=1 TO edge
1070   t=e*ph*2
1080   DRAWR size*COS(t),size*SIN(t)
1090   NEXT
1100 MOVE h,v
1101 IF flag%>0 THEN FILL flag%
1110 RETURN
1120 :
```

Exercise 8.3 Text-screen dump

```
1 EVERY 128 GOSUB 9000
10 REM ********************************
11 REM ** Exercise  8.3 :             **
12 REM ** TEXT-SCREEN DUMP            **
15 REM ********************************
60 :
9000 REM -- interrupt-driven text screen dump:
9010 IF VPOS(#0) <= 16 THEN RETURN
9020 REM -- only after 16 lines.
9030 vvvv=VPOS(#0)
9050 FOR r8%=1 TO vvvv
9060 FOR c8%=1 TO 40
9070 LOCATE c8%,r8%
9080 PRINT #8, COPYCHR$(#0);
9090 NEXT c8%
9100 PRINT #8
9110 NEXT r8%
9120 CLS: RETURN
9125 REM set for 40-column mode.
9130 :
```

Index of example programs

Index

The default INK settings

Inkpot	Colours		
	Mode 0	Mode 1	Mode 2
0	1	1	1
1	24	24	24
2	20	20	1
3	6	6	24
4	26	1	1
5	0	24	24
6	2	20	1
7	8	6	24
8	10	1	1
9	12	24	24
10	14	20	1
11	16	6	24
12	18	1	1
13	22	24	24
14	F1,24	20	1
15	F16,11	6	24

You can change the colour in each inkpot to any that you like. It doesn't matter what the particular colours are but you are only allowed two in MODE 2, four in MODE 1 and sixteen in MODE 0.

The Amstrad's colour codes

0	Black	14	Pastel Blue
1	Blue	15	Orange
2	Bright Blue	16	Pink
3	Red	17	Pastel Magenta
4	Magenta	18	Bright Green
5	Mauve	19	Sea Green
6	Bright Red	20	Bright Cyan
7	Purple	21	Lime Green
8	Bright Magenta	22	Pastel Green
9	Green	23	Pastel Cyan
10	Cyan	24	Bright Yellow
11	Sky Blue	25	Pastel Yellow
12	Yellow	26	Bright White
13	White		

Richard Forsyth and Brian Morris

The AMSTRAD BASIC Idea

Amstrad have produced a remarkable range of computers. Nevertheless, many owners are unable to unleash their full potential. Why? Chiefly because they have not made the transition from computer user to computer programmer.

The aim of this book is to ease that transition. It will enable you to use AMSTRAD BASIC to solve realistic problems. In other words, it will turn you from a passive user into an active one – no longer dependent solely on pre-packaged programs but also able to explore the full potential of your machine.

This book – the successor to the best-selling *The BBC BASIC Idea* – will help you to become a competent programmer in AMSTRAD BASIC. It shows in a straightforward manner how to use modern methods of problem analysis and design to solve your programming problems, so that you can enjoy and profit from your computing.

Richard Forsyth is an author who also runs his own business, Warm Boot Limited, which specializes in machine-intelligence applications.

Brian Morris is a freelance writer and software consultant, who specializes in robotics.

Cover design based on an idea by Julian Dorr.

A Chapman and Hall/
Methuen Paperback
COMPUTING

11 New Fetter Lane,
London EC4P 4EE
29 West 35th Street,
New York NY 10001

ISBN 0-412-28070-1

9 780412 280702

The AMSTRAD BASIC Idea Forsyth CHAPMAN AND HALL/METHUEN

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/